HARDWARE ACCELERATION FOR POWER EFFICIENT DEEP PACKET INSPECTION

Yachao Zhou, MSc.

A Dissertation submitted in fulfillment of the requirements for the award of Doctor of Philosophy (Ph.D.)

Dublin City University

School of Electronic Engineering

Supervisor: Dr. Xiaojun Wang

August 2012

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:

ID No.:

Date:

ACKNOWLEDGMENTS

Firstly, I would like to thank my supervisor Dr. Xiaojun Wang for giving me guidance and support during my time in Dublin City University. I would also like to thank Dr. Olga Ormond and all my colleagues in Network Processing Group (NPG) and Network Innovations Centre (NIC) Institute for their assistance with my work. I would like to thank Dr. Tian Song and other members in his group in Beijing Institute of Technology. Through heuristic discussions, he helps me find a flavour of joy in doing research and build a rigorous way of thinking when writing a paper.

I am thankful to Dr. Noel Murphy for giving me the opportunity and advices of lecturing. My gratitude also goes to staff members in School of Electronic Engineering for their kind help during my study. I would also like to thank my master supervisor Prof. Bin Liu, Prof. Chengchen Hu and all my teachers and friends in Beijing University of Posts and Telecommunications (BUPT) and Tsinghua University; I am blessed to have been with them.

I would like to remember all my friends in Ireland who have accompanied me over these years. They have different interests but they all bring me a positive and enthusiasm attitude towards life. Also with their companion, I saw the beautiful views around Dublin.

Finally, I would like to give my deepest thanks to my family by giving me firm support over the years. Because of their confidence in me, it is well worth all the effort involved in the completion of this work. Special thanks to my husband, a happy marriage is an earlier heaven, you are wonderful and we always keep love and appreciation in mind for each other.

ABSTRACT

The rapid growth of the Internet leads to a massive spread of malicious attacks like viruses and malwares, making the safety of online activity a major concern. The use of Network Intrusion Detection Systems (NIDS) is an effective method to safeguard the Internet. One key procedure in NIDS is Deep Packet Inspection (DPI). DPI can examine the contents of a packet and take actions on the packets based on predefined rules. In this thesis, DPI is mainly discussed in the context of security applications. However, DPI can also be used for bandwidth management and network surveillance.

DPI inspects the whole packet payload, and due to this and the complexity of the inspection rules, DPI algorithms consume significant amounts of resources including time, memory and energy. The aim of this thesis is to design hardware accelerated methods for memory and energy efficient high-speed DPI.

The patterns in packet payloads, especially complex patterns, can be efficiently represented by regular expressions, which can be translated by the use of Deterministic Finite Automata (DFA). DFA algorithms are fast but consume very large amounts of memory with certain kinds of regular expressions. In this thesis, memory efficient algorithms are proposed based on the transition compressions of the DFAs.

In this work, Bloom filters are used to implement DPI on an FPGA for hardware acceleration with the design of a parallel architecture. Furthermore, devoted at a balance of power and performance, an energy efficient adaptive Bloom filter is designed with the capability of adjusting the number of active hash functions according to current workload. In addition, a method is given for implementation on both two-stage and multi-stage platforms. Nevertheless, false positive rates still

prevents the Bloom filter from extensive utilization; a cache-based counting Bloom filter is presented in this work to get rid of the false positives for fast and precise matching.

Finally, in future work, in order to estimate the effect of power savings, models will be built for routers and DPI, which will also analyze the latency impact of dynamic frequency adaption to current traffic. Besides, a low power DPI system will be designed with a single or multiple DPI engines. Results and evaluation of the low power DPI model and system will be produced in future.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

LIST OF ACRONYMS

ISP	Internet Service Providers
NIDS	Network Intrusion Detection System
DPI	Deep Packet Inspection
DFA	Deterministic Finite Automata
NFA	Nondeterministic Finite Automata
AC	Aho-Corasic
CDFA	Cached Deterministic Finite Automate
CBF	Counting Bloom Filters
SBF	Spectral Bloom Filters
DBF	Dynamic Bloom Filters
FPGA	Field-Programmable Gate Array
TCAM	Ternary Content-Addressable Memory
NP	Network Processor
ACU	Adaptive Clock Unit
PE	Processing Units
FIFO	First In First Out
PCB	Printed Circuit Board
DFVS	Dynamic Frequency and Voltage Scaling
VCD	Value Change Dump

PUBLICATIONS

Journals

Y.C. Zhou, Y. Tang, X.J. Wang, "Tag-DFA for Improved DFA Compression for Deep Packet Inspection", *Elsevier, Procedia Engineering*, ISSN:1877-7058, Volume 29, pp.3755-3762, 2012.

Y.C. Zhou, T. Song, W.L. Fu, X.J. Wang, "C^2BF: Cache-based Counting Bloom filter for Precise Matching in Network Packet Processing", *Elsevier, Procedia Engineering*, ISSN:1877-7058,. Volume 29, pp.3747-3754, 2012.

Y. C. Zhou, X.J. Wang, "Memory Efficient Deep Packet Inspection for Application Detection in Mobile Internet", *Journal of Harbin Institute of Technology*, ISSN:1005-9113, Volume 18, Issue SUPPL. 1, pp. 398-402, December 2011.

Conferences

Y.C. Zhou, T. Song, X.J. Wang, "EABF: Energy Efficient Self-Adaptive Bloom Filter for Network Packet Processing", *IEEE International Conference on Communications (ICC)*, Ottawa, Canada, June 10-15, 2012.

Y.C. Zhou, X.J. Wang, "An Improved Implementation of Montgomery Algorithm Using Efficient Pipelining and Structured Parallelism Techniques", *the 21st IET Irish Signals and Systems Conference* (*ISSC 2010*), UCC, June 23-24, 2010, Cork, Ireland.

Y.C. Zhou, X.J. Wang, "Efficient Pattern Matching with Counting Bloom Filter", *China-Ireland International Conference on Information and Communications Technologies (CIICT 2010)*, Oct 10-11, 2010, Wuhan, China.

Y.C. Zhou, X.F. Wang, Y. Tang, J.Y. Yang, X.J. Wang, "JavaScript code Fragment Analyzing for Cache Proxy", *China-Ireland International Conference on Information and Communications Technologies (CIICT 2009)*, Aug 19-21, 2009, Maynooth, Ireland.

X.F. Wang, C.C. Hu, **Y.C. Zhou**, X.J. Wang, B. Liu, "Efficient Log System for Distribution Behavior Analysis in Peer-to-Peer Networks", *China-Ireland International Conference on Information and Communications Technologies (CIICT 2010)*, Oct 10-11, 2010, Wuhan, China.

K.Q. He, C.C. Hu, J.C. Jiang, **Y.C. Zhou** and B. Liu, "A2C: Anti-Attack Counters For Traffic Measurement", *Proceeding of IEEE GLOBECOM*, December 6-12, 2010, Miami, Florid.

C.H. Sun, C.C. Hu, **Y.C. Zhou**, X. Xin and B. Liu, "A More Accurate Scheme to Detect SYN Flood Attacks", in *Proc. IEEE INFOCOM Student Workshop*, Rio de Janeiro, Brazil, Apr. 19-25, 2009.

Chapter 1: Introduction

As Internet traffic loads grow, with higher and higher numbers of mission critical services and high customer expectations for anytime anywhere service, the need for good data security is increasingly significant. NIDS is an effective way to safeguard the Internet and one key procedure in NIDS is DPI. This chapter introduces why DPI is important for network security and how DPI works. After explaining current problems of DPI, the objectives and contributions of this thesis are also presented.

1.1 Network Trends

The Internet penetration is growing at a fast rate. This has been brought about by the wide variety of access devices, such as desktops, laptops, tablets, netbooks, and smart phones, reasonable rates for data bundles from wired and wireless network operators, and the ease of accessible information and variety of innovative applications. Almost anyone can access the 'global village' of the World Wide Web with very little investment. Social networking, blogging, and easy to design web-pages have made having a global web presence very accessible for people of varying technical ability and budget levels, both for personal and business use, and from any corner of the world. Examples of popular applications include: Bittorrent, Youtube, Skype, eBay, Internet Banking, and Online Booking. These applications have made life more convenient for many people who have become accustomed to a good user quality of experience and now rely on 24/7 service availability.

The growth of the Internet reflects in the rapid increase of the population. The Internet World Stats survey [1] shows the Internet penetration has reached 32.7% of the world's population as of December, 2011, with the number of Internet users growing by 141.7 times from December, 1995 in the past 16 years, and growing by 1.52 times from December, 2006 in recent 5 years. The growing population of the Internet users generates large traffic volumes and requires high network bandwidth. In

future years, the growth will remain strong considering the potential users in China, India and other developing countries. Up to now, 44.8% of Internet users are in Asia with a penetration rate of 26.2%, compared to a penetration rate of 41.0% in the rest of the world [1]. Based on data derived from Cisco Visual Networking Index (VNI) Global forecast [2], the traffic amount will grow at a compound annual growth rate (CAGR) of 34% from 2009 to 2014, reaching about 75% of a Zettabyte per month by 2014 [2].

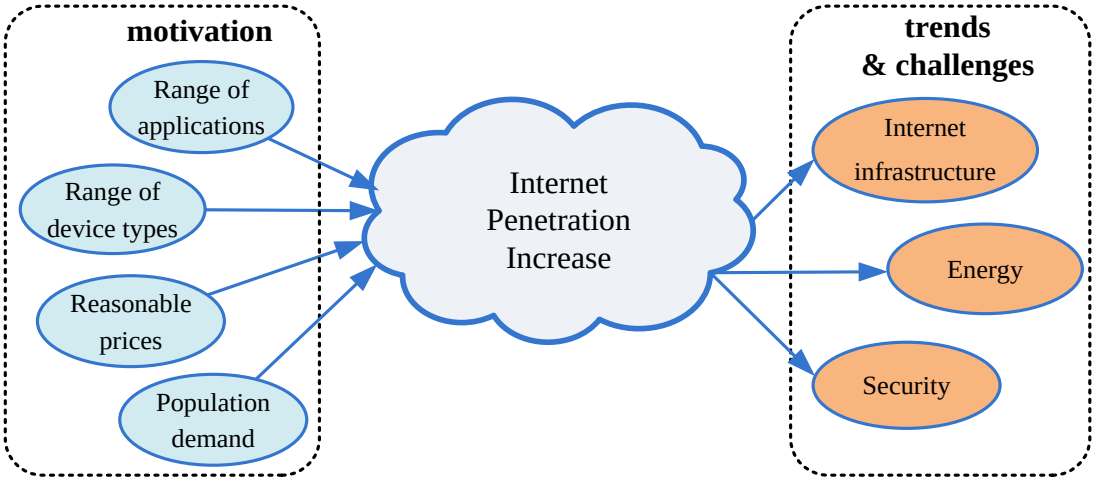


Figure 1-1 Aspects of the Internet growth

Extrapolating from current Internet trends and Next Generation Network (NGN) projects, it is possible to speculate the following trends and challenges with the Internet development: 1) the Internet infrastructure, 2) network security and 3) energy efficiency, as shown in Figure 1-1.

1) Internet infrastructure

The Internet infrastructure allows self-evolution without a central controller, enabling the interconnection of diverse applications over heterogeneous networks, e.g. access from home/office networks, public hotspots, cellular network or wireless network etc. running at a wide range of speeds. Bandwidth consumption is not evenly distributed. The top 10% of connections is responsible for over 60% of worldwide broadband Internet traffic [3]. Open source platforms and programs have also accelerated the rate of new service offerings and innovative Internet applications and

services [3], and in addition, their associated volume of data traffic continues to increase.

Ultra-high bandwidth systems deployed at home and in the office are making bandwidth consuming applications more popular. Network traffic for Internet video, file sharing, video calling, online gaming etc. has seen massive growth. Based on data in [2], Internet video has replaced Peer-to-Peer (P2P) as the top traffic type, approaching 40% at the end of 2010, which is the first time that P2P traffic was not the largest Internet traffic type since 2000. P2P is also growing in volume, but declining in its percentage of overall IP traffic. Internet video alone will account for 57% of Internet traffic in 2014 [2].

Mobile Internet has become a very attractive business with extremely fast growing speed and great market potential all over the world. A wide range of mobile devices drive the rapid growth of the mobile Internet. Statistics in [4] show that more than 1 billion of over 4 billion mobiles phones worldwide are now smart phones (i.e. over 27%), and 3 billion are SMS enabled (i.e. 75%). In 2011, more than 50% of all "local" searches are done from a mobile device. By 2014, mobile Internet usage will overtake desktop Internet usage [4]. In particular, over 200 million Facebook users (that is, one third of all Facebook users) access Facebook from a mobile device and 91% of all mobile Internet usage is "social" related [4]. Correspondingly, the Internet adapts network processing and applications to mobile environment.

2) Energy consumption

The increase in network energy consumption has become a challenge for the Internet. To keep pace with the Internet and its related services for an ever-growing user population, the maximum data throughput rate of core network routers and switches needs to increase correspondingly. While the performance improvement of integrated circuits roughly follows Moore's law, the Internet growth speed has gone beyond Moore's law over recent years. Networks that were once over-provisioned to handle high performance even at busy hour traffic loads need to be made even more

efficient to cope with higher loads. One solution for network servers to cope with handling increasing peak traffic loads is through redundancy designs with multi-core or multi-rack. For example, multi-rack architectures in high-end IP routers increase router capacities by a factor of 2.5 every 18 months [5], which is faster than Moore's law. On the other hand, as suggested by Denard's scaling law [6], silicon technologies improve their energy efficiency with a lower rate compared to traffic volume and routers' capacity increase. As a result, the energy savings cannot make up the increase of Internet energy consumption.

Meanwhile, energy bills for network operation are an added limitation especially with increasing pressure to meet CO_2 reduction targets. It is only in recent years that Internet Service Providers (ISPs) have begun to have raised awareness of the need for more energy efficient wired networks and service infrastructures. As energy prices and security of energy supply becomes a bigger issue, energy is becoming a key network performance indicator. ISP's are caught in a struggle to trade-off between the need for a reduction in their energy bill and the requirements to increase the number of devices with sophisticated architectures to perform increasingly fast and complex processing.

Future Internet requires the re-thinking of network infrastructure towards energy efficiency. The total energy consumed by Internet related devices is climbing fast. The datacenters, servers, links, routers and other network devices run constantly and consume large amounts of energy. Towards the green Internet, energy efficiency of cyber-infrastructure and network applications is high on the agenda, which can help to create a low carbon society. Traditional network targets lower cost and better performance, but the objectives of next generation networks also include low energy consumption. Generally, low power techniques exploit traffic fluctuation on network links. Typical Ethernet traffic profiles have low average link utilization over time with occasional bursts of network activity. The energy spent during idle link periods can be reduced with a lower frequency or voltage.

3) Security

While the Internet has brought great convenience to everyday life, it is by no means a safe place. The Internet is successful because it makes almost all Internet resources available to all users simultaneously. This resource sharing ability is central to the Internet's utility and success. On the other hand, the Internet is constructed with simple cores and complex end hosts. For fast speed line-rate processing, core routers have been designed to be as simple as possible with only routing and forwarding functions.

With the growing numbers of users and personal data online, the Internet has become more and more vulnerable to various attacks. The latest Internet security report published by Symantec [7] shows a massive increase of 81% in malicious attacks over the previous year. Fraudulent phishing causes enormous economic loss, and hackers' sniffing leads to an immeasurable leakage of personal information. More than 232.4 million identities were exposed overall during 2011 [7].

For the mobile Internet, the worldwide sales of smart phones reached 144.4 million units in the first quarter of 2012, according to the analyst firm Gartner [8]. The increase in the use of smart phones leads to an increase of online attacks if mobile data is not properly protected. The security threat report from Symantec [7] said mobile vulnerabilities increased by 93% in 2011, which was the first year that mobile malware presented a tangible threat to businesses and consumers.

Another major application trend coming in the near future is smart grid. Security of personal information especially from smart meters in people's home will be important. Therefore, new cyber security products targeted at smart grids are needed. The report data [7] also showed that targeted threats are not limited to the Enterprises and executive level personnel, but also to businesses of all sizes, no one is immune to attacks.

Traditional Internet technologies need to be renovated to meet the new demands of both ISPs and consumers.

1.2 Motivation

The work in this thesis focuses on enhancing network security using Deep Packet Inspection (DPI) in Network Intrusion Detection System (NIDS), improving DPI performance through hardware acceleration, while also optimizing the power efficiency.

The security challenges in NGN networks mainly come from three aspects: firstly, the number of security incidents continues to increase exponentially; secondly, the complexity and sophistication of attacks and vulnerabilities continue to rise; and thirdly, the potential impact of security threats to the network is getting more and more significant.

NIDS is a powerful tool to identify potential malicious attacks in networks, enhancing the safety of core networks. The main components of NIDS are packet classification and DPI. In general, packet classification checks specific fields of packet header, and DPI inspects on the whole packet payload against predefined patterns. While the packet header follows a uniform format, the positions where the patterns might appear in the packet payload are unpredictable, thus the entire packet payload needs to be searched. For example in traditional networking applications, packet classification only analyzes the packet header, the equivalent of looking at the address on an envelope; on the other hand, opening the envelope and reviewing the contents (i.e. packet payloads) are the work of DPI.

Since inspection on packet headers is no longer enough to detect malicious traffic, in recent years, DPI is beginning to play a more significant role in network security and network management. Based on DPI methods, NIDS can be classified into two categories: 1) anomaly-based detection system and 2) pattern matching based detection system.

Anomaly-based system looks for unusual behavior using statistical flow analysis methods. Referring to *experiential* defined anomaly behaviors, an anomaly-based NIDS is able to detect zero-day attacks as soon as they occur, including attacks that

are previously unknown and or modifications of well-known attacks. In an anomaly-based NIDS, a model of normal data or usage patterns is built at the training stage; afterwards, the data and usage patterns of incoming packet payloads, which significantly deviate from the normal data and usage patterns, will be marked as anomaly. Although having existed for a long time, anomaly-based systems are mostly limited to academia at the moment. This is mainly due to the challenges in reasonable determinations of “normal” activities. As a result of these challenges, anomaly-based systems still produce large false positives and false negatives.

Detection systems based on pattern matching inspect packet contents against a set of previously extracted patterns including signatures and rules to identify certain viruses, applications or network protocols. Signatures explicitly define what kinds of activities should be considered malicious in the form of fixed strings or regular expressions. Rules define more constraints on packet payloads in addition to those on IP addresses, ports and protocols. The majority of IDS commercial products belong to the pattern based category. Snort [9] and Bro [10] are popular NIDS systems used in the network layer and L7-filter [11] is commonly used for application layer protocol analysis and traffic classifications in the application layer.

In addition to its application for network security, DPI is also used for other purposes such as Internet censorship of sensitive contents within certain domains. For example, L7-filter [11] can be used for the higher level monitoring and censoring of traffic content, detects applications such as online media service, file-downloading or Voice over Internet Protocol (VoIP) phone calls. This identification of different protocols can be used for network management provisions such as providing different Qualities of Service (QoS) levels to different applications or streams if necessary. Moreover, some application traffic is routed based upon its content. For instance, all video traffic originating from Google can be identified with specific patterns and then routed according to service level agreement.

DPI methods include fixed string based pattern matching and regular expression based pattern matching, as shown in Figure 1-2. Basically, DPI works by comparing

incoming packets byte by byte against predefined patterns to identify specific viruses, attacks and protocols. Fixed string matching algorithms such as Bloom filter etc. are fast and memory efficient. Regular expression matching algorithms, such as Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA), have much larger computation and storage complexities. They are more widely used in recent years since regular expression provides much better flexibility in representing the ever-increasing signature dataset.

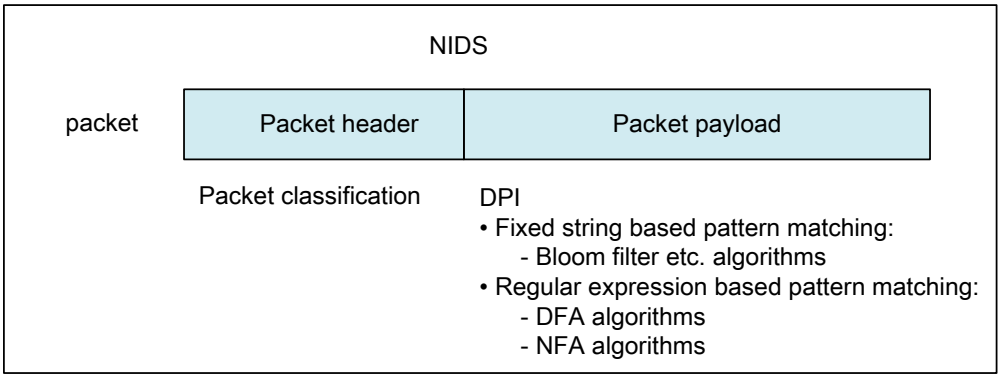


Figure 1-2 DPI and packet classification in NIDS

Bloom filter [12] with a group of hash functions is an effective DPI solution, where the incoming traffic are mapped to compare with trained hash tables generated from fixed string DPI patterns. Hash functions essentially map from a large and irregular data set to a compact and regular data set. Despite the excellent average-case performance, Bloom filter results include false positives and require additional verification. There are some improved Bloom filter structures, such as Counting Bloom filter, Spectral Bloom filter and Dynamic Bloom filter etc., which are more memory efficient or have smaller false positive probability. In addition, Bloom filter based approaches are ideally suitable for hardware implementation.

DFA and NFA algorithms are two main approaches to perform DPI with regular expression matching. NFAs have smaller size, since the number of states in an NFA is usually comparable to the number of characters presented in its regular expressions. But NFAs requires high and unpredictable memory bandwidth. On the other hand, DFA works fast by proceeding one character each clock cycle. But DFA has the

potential state explosion problem, which may result in extraordinarily large DFA.

Deployed at the nodes of core networks, DPI algorithms are required to be time and space efficient, so that they can scan packet payloads in routers at wire-speed. A number of researches on DPI improve its matching speed and reduce its memory consumption. Moreover, their pattern databases also need to update quickly to deal with new emerging Internet conditions.

Hardware-based accelerators with parallel processing engines are utilized for high speed pattern matching. Due to the complexity of rule sets and the necessity to scan the whole payload, software implementation is unable to keep up with network wire speed. The commonly used hardware platforms for pattern matching include Ternary Content-Addressable Memory (TCAM), Field-Programmable Gate Arrays (FPGA) and Network Processors (NP). Comparing TCAM and FPGA, TCAM is specially built for parallel comparison and more suitable for simpler tasks and FPGA is more flexible for more complex tasks. On the other hand, TCAM is cheaper and can run at higher frequency than FPGA at the cost of high energy consumption. NP is more convenient for applications than FPGA, but the size of memory that can be assigned for each processing unit is still too limited to store all the pattern strings.

Furthermore, the energy efficiency of network processing is getting more and more attention from both academia and industry. Energy consumption of idle or underutilized components can be reduced through dynamic power management, which is adaptive to traffic load. Low power techniques such as clock gating, power gating, and dynamic frequency control can be applied at component, system and network levels. Under certain QoS constraints, low power DPI systems are designed to save power as much as possible.

1.3 Research Goals

DPI on packet payload plays an important role in both network security and traffic monitoring applications. Given a set of previously extracted patterns, packet

payloads are matched to identify the predefined patterns, protocols or network applications. In order to identify network packets at wire speed, the pattern matching algorithms are required to be time and space efficient and the signature rule sets can also be updated conveniently.

There are two basic assumptions used widely for researches on DPI. Firstly, it is assumed that the DPI signatures are known in advance. The signatures are gathered over real traffic flows by specialized hardware and software in network routers, for industry or research use. The signature databases are updated frequently with new emerging traffic features. For example, ClamAV is an open source anti-virus toolkit [13]. ClamAV Virus Database reports new viruses which are not detected by ClamAV and clean files which are incorrectly detected by ClamAV. In this thesis, the patterns to be matched against packet payloads are extracted from the rule sets in Snort [9], Bro [10] and Linux L7-filter [11]. The second common DPI assumption is that the payload data to be scanned is the continuous payload data after packet reassembly. The packet payloads in continuous packets are contents fragments and may be out of order. TCP reassembly reconstructs the correct application data at the end host. Typical speeds reported for existing packet reassembly systems are 1 to 10 Gbps under normal traffic [14], which is fast enough for DPI.

The incoming pressure for handling higher volume of data at faster speed with resource efficiency brings harder requirements for DPI. Deployed at network routers, DPI is required to work at high speed without an impacting the demanded speed of packet storing and forwarding. Moreover, the memory requirements should be within the capacity and budget of routers. To keep up with growing traffic rate and the size of rule set, DPI should satisfy the following requirements: 1) high throughput at wire-speed; 2) high memory efficiency especially for hardware designs; 3) the wire-speed in 1) should guarantee the worst case performance, so that DPI is capable of working under all kinds of network conditions; 4) capability of on-line signature update.

The overall goal of this thesis work is to design energy efficient DPI algorithms

for parallel hardware implementation. The objectives can be summarized as follows.

- The first research objective is to improve the memory efficiency of DFA algorithm. DFA is the most popular algorithm in the DPI system. Traditional DFA based regular expression matching algorithms can be optimized to reduce memory redundancy in DFA states and transitions.
- The second objective is to speed up the pattern matching process. Pre-filter based methods are effective and practical, since the inspection on network traffic is normally performed using a very small percentage of the whole matching machine.
- The third objective is to develop hardware level DPI algorithms. Currently, software level DPI algorithms cannot keep up with the growth of backbone network bandwidth. A parallel Bloom filter architecture can be used for fast DPI. Another goal for Bloom filters on FPGA platforms is to create implementation that supports easy and fast pattern updates.
- The fourth objective is to reduce power consumption of the DPI systems. There is an increased need to be more energy aware. Energy efficiency can be achieved by using adaptive voltages or frequencies according to system workload. With QoS constraints satisfied, the essence of existing energy aware approaches is to dynamically turn down or switch off network functional components during periods of light utilization.

1.4 Contributions

The main contributions of this thesis can be summarized as follows:

- Designed two memory efficient DFA algorithms, Extended-D^2FA and Tag-DFA;
- Developed a Counting Bloom Filter (CBF) based parallel architecture for fast DPI implementation;

- Designed an Energy efficient adaptive Bloom filter (EABF) algorithm, based on two-stage and multi-stage platforms, targeted at an adaptive balance of performance and power consumption based on current traffic load;
- Developed a fast precise matching method on FPGA, using a combination of CBF and cache, C^2BF;

1.5 Thesis Organization

The remainder of this thesis is structured as follows:

- *Chapter 2: Research Background*

  The background chapter introduces an overview of DPI algorithms and also Bloom filter algorithms aimed for efficient hardware implementations. This chapter first introduces the basic and improvements of fixed string pattern matching methods, the DFA and NFA based regular expression matching. Towards hardware implementation, Bloom filter and its variations are presented. Low power techniques are also shown for power efficient DPI system design.

- *Chapter 3: Memory Reduction of DFA Algorithms*

  This chapter analyzes the redundancies in DFA transitions for compression. Two improved DFA methods, Extend-D^2FA and Tag-DFA, are proposed to exploit more than one kind of redundancy to compress DFA transitions, and are compared with a well-known algorithm D^2FA.

- *Chapter 4: Extended Bloom Filter for Network Packet Matching*

  This chapter first presents multi-pattern matching using multiple parallel Bloom filters. With pruned and list-balanced Counting Bloom Filter (CBF), the Bloom filter based pattern matching system consumes much less memory in FPGA implementation compared with that of the original scheme. Second, an energy efficient adaptive Bloom filter model EABF is proposed for a

balance of power efficiency and performance. EABF algorithm, based on two-stage and multi-stage platforms, adjusts the working stage of hash functions in a Bloom filter. Thus the adaptive Bloom filter maintains a dynamic balance of power and processing speed. Third, for feasible precise matching, the Bloom filter is combined with cache mechanism aimed at larger cache hit rate.

- *Chapter 5: Future Work Proposal - Power Modeling and Low Power DPI*

  This chapter is the proposed future work of power models and the methods for low power DPI. Except for the preliminary results for dynamic frequency scaling, other results for the work in this chapter are not available since they are still limited to the models and designs of the algorithms.

- *Chapter 6: Conclusions and Future Work*

  A summary of the contributions made in this thesis and the research objectives attained are presented, as are plans for future work.

Chapter 2: Research Background

This chapter introduces the research and technical background for the work presented in this thesis. It starts with an overview of DPI algorithms and introduces the basic and more improved versions of fixed string pattern matching methods; both the DFA and NFA based regular expression matching methods. In terms of hardware implementation, the Bloom filter technique and its variations are presented. Low power techniques for power efficient DPI system design are also presented.

2.1 DPI in NIDS

The Internet, despite its huge success, has also spawned a new generation of computer crime or cybercrime. Cybercrime covers two categories: 1) crimes that target computer or network devices directly; and 2) crimes that target application users and data. The first category includes computer viruses, denial-of-service attacks and malware attacks, using specially designed programs to disrupt normal operations. The second category includes fraud, identity theft and phishing scams. There have been lots of reports on bank fraud and theft of classified information, which seriously threatens information security and financial health. Many victims are not aware of privacy loss. Investigation against Internet fraud ring reveals millions of unknowingly affected cases worldwide. As reported by Symantec, the number of unique malware variants increased to 403 million in 2011, compared to 286 million in 2010, and the number of web attacks detected per day has increased by 36% [7].

People are building their lives around wired and wireless networks, and therefore there is a strong need to make every effort to defend the Internet. The intrusion detection system (IDS) can be deployed as a network-based sensor, host-based agents or a specialized security management and monitoring network, as shown in Figure 2-1.

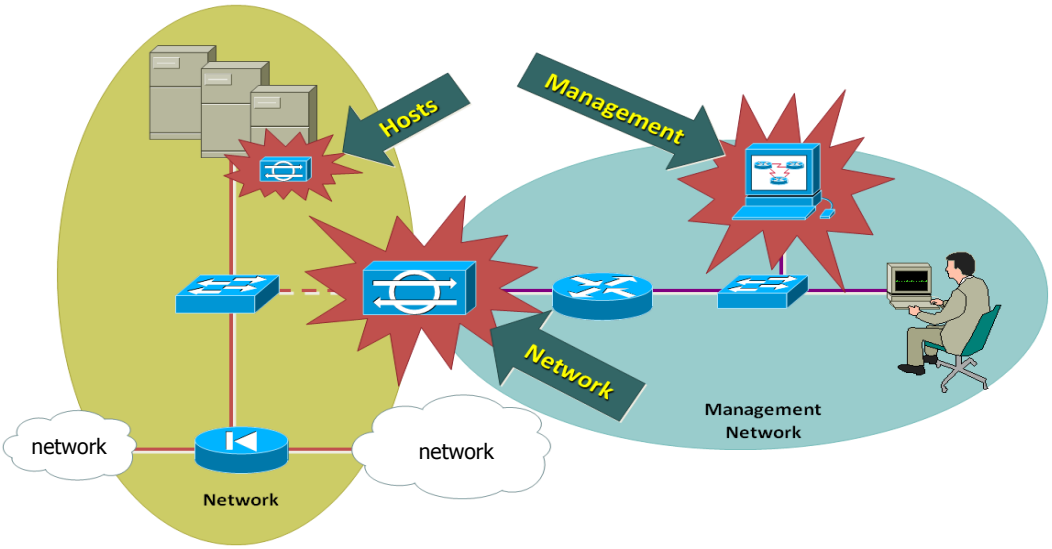


Figure 2-1 Deployment of IDS system

When a NIDS is deployed on a host, its maintenance cost is high due to the need for frequent updates. For network-based deployments, NIDS can be set up between the un-trusted external Internet and the trusted internal network, either before or behind firewalls, as shown in Figure 2-2.

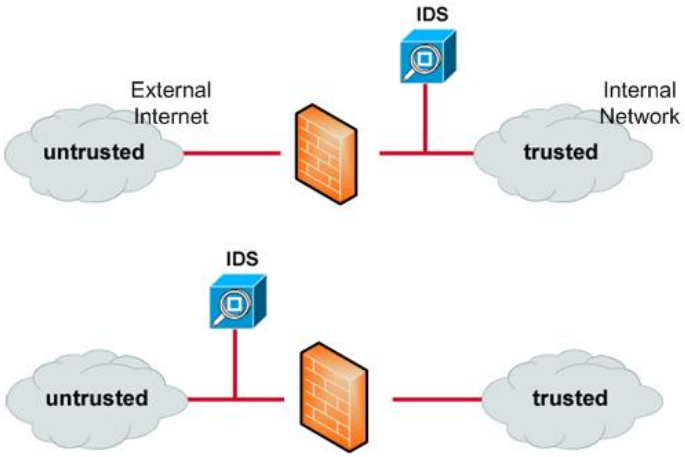


Figure 2-2 NIDS deployments in network

NIDS is an efficient and effective way to safeguard network security. A key component in NIDS is its DPI module, which scans network packets byte after byte to match the patterns defined in the local security database. The security risks and new network vulnerabilities are updated every day.

Based on DPI implementation, NIDS can be classified as an anomaly-based detection system and a signature-based detection system. The main difference

between the two kinds of technologies represents in the concepts of “anomaly” and “attack” [15]. An anomaly is usually an event that is suspicious from the perspective of security. An attack is a sequence of operations that threaten system security, where patterns can be summarized as the signatures of these operations. This distinction determines different characteristics of the two systems. Anomaly-based detection system is capable of detecting new and unknown intrusions. On the other hand, signature-based detection system is good at identifying well-known attacks that have been specified.

- **Anomaly-based detection systems**

Anomaly-based detection technique is a valuable technology to protect network against malicious activities, especially new types of attacks. Anomaly-based detection systems estimate normal behaviours of the system or network and generate alert whenever the observation gets away from the normal behaviours. In general terms, anomaly-based detection systems consist of three basic stages: ***parameterization stage***, ***training stage*** and ***detection stage*** [15]. Firstly, the instances of the system are represented. Secondly, the normal and abnormal behaviour of the system is characterized and a model is built. Thirdly, the model is compared with the parameterized observed traffic, and it raises an alarm whenever the deviation exceeds a given threshold.

Compared to signature-based detection system, anomaly detection system can detect zero day attacks but it tends to have a high false positive rate that identifies non-malicious packets as malicious packets. As a result, anomaly detection technique is not popularly used in industry. A few security vendors are just starting to claim that their systems include anomaly detection in their detection engines.

- **Signature-based detection system**

Signature-based detection system matches the packet contents against pre-defined patterns and is the mainstream NIDS approach. This thesis focuses on pattern matching methods. Certain kinds of patterns in packet payloads, which are

considered as representatives of attacks, are summarized by vendors of security devices or network experts.

Signature-based DPI depends on using either fixed string matching or regular expression matching. Fixed string matching, e.g. Bloom filters, is fast and suitable to implement on industrial security devices. As security databases grow increasingly more complicated, the number of DPI update operations grows correspondingly. Nowadays, regular expression matching has become more popular due to its flexibility and rich expressiveness. Though flexible in representing security threats, regular expressions based approaches, e.g. NFA and DFA, have significant computation and storage complexities. State machines with states and transitions are built by NFA and DFA algorithms and matched against packet payloads.

An NFA is a finite state machine where from each state and a given input, the automaton jumps to several possible next states, while the automaton of a DFA jumps to a uniquely determined next state. Given a set of patterns, they are first expressed using an NFA that follows the symbols in the patterns; and if necessary, the NFA can be translated to an equivalent DFA using the powerset construction. Practically, it is useful to convert the easier-to-construct NFAs to the more executable DFAs.

An example of DFA and NFA is shown in Figure 2-3. An NFA can represent regular expressions with a smaller size, since its number of states is of the same order as the number of characters in regular expressions. But an NFA has multiple active states concurrently, so that processing a single character may incur a number of state traversals, which leads to unpredictable memory bandwidth costs. A DFA has at most one active state at any time during the entire matching process and needs only one state traversal to process one character. However, the deterministic bandwidth is at the cost of a larger memory requirement, since some expressions might incur state explosions when an NFA is transformed to a DFA.

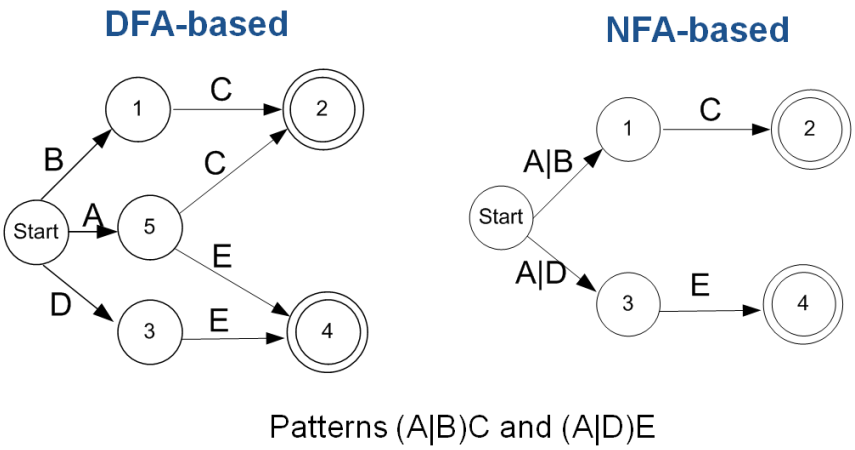


Figure 2-3 Examples of DFA and NFA automata

In terms of time and memory efficiency, researches on regular expressions mainly focus on two aspects. The first technology is to minimize the number of states and transitions with compression techniques, including default transition compression [16], run-length encoding, alphabet reduction [17] and state merging [18] etc. The second technology is to develop a novel kind of state machine in addition to basic DFA and NFA, such as CD^{2}FA [19], δFA [20], Hybrid-FAs [21] and XFAs [22] etc. which aims at easing the drawbacks of NFAs and DFAs while keeping their strengths.

Network applications attach great importance to speed and are required to guarantee the worst-case performance. Software-based DPI algorithm is inevitable a bottleneck in NIDS. Researchers resort to hardware accelerators on FPGA, TCAM or network processors that enable parallelism for high speed packet processing. Bloom filter is suitable for hardware implementation of DPI.

2.2 DPI Algorithms

2.2.1 DPI Patterns

Essentially, there are three types of patterns in DPI rule sets including single patterns, composite string patterns and regular expressions. DPI matches these patterns with fixed string matching or regular expression matching.

- **Single patterns**

Single patterns are patterns with a fixed number of characters.

- **Composite patterns**

Composite patterns correlate single patterns through "negation", "and", "or" etc. operations.

- **Regular expressions**

Instead of enumerating all the strings explicitly, regular expressions describe a set of strings with signature symbols, such as character sets, repetition of characters and counters.

Character sets have two forms: 1) range sets with lower and upper limits (like "[c_{low}-c_{up}]"), and 2) cumulative symbols. Cumulative symbols include wildcard ("*") representing any character, space characters ("\s"), all digits ("\d"), all alpha-numerical characters ("\w"), and their complementary sets ("\S, \D, \W"). And symbol "^" to represent a complementary set of the next character, for example, "^*a*" represent a set of all the characters except *a*. Since character sets cannot be directly stated in DFA, a solution is to use exhaustive enumeration of all the exact-matching strings.

The implementations for the repetitions of simple characters (like "c^{+}" and "c^{*}") or character sets (like "$(\w)^{+}$" or "$.^{*}$"), consume large amounts of memory and are difficult to be compressed. Since the repetitions cannot be converted to an exact form of strings, some DPI algorithms like hashing or bit-split techniques are not able to efficiently represent these repetitions. Moreover, repetitions of character sets and wildcard will cause states explosion of a DFA [17]. A common solution to this problem is to use multiple-DFAs instead of a single DFA.

Counting constraints are used to specify the number of repetitions, with or without an upper bound. Counters could also cause state blow-up as explained in [17] with counters on large character sets or wildcards.

2.2.2 Fixed String Matching Algorithms

Fixed string DPI algorithms include single pattern matching and multi-pattern matching algorithms.

Boyer-Moore [23] and KMP [24] algorithms are two widely used single-pattern matching algorithms, which search for one pattern each time. KMP [24] matches fast with pre-computed table that instructs the next character in the pattern to be compared with, or the next position to be moved to, after a comparison failure. Baker et al. in [25] decrease the maximum complexity of KMP, and implement a hardware-based DPI architecture with parallel KMP algorithm. Boyer-Moore [23] builds two tables over the pattern indicating how many characters to shift forward and backward, so that it doesn't need to check every character in the matching pattern. To make it simpler, Horspool algorithm [26] trades space for time to obtain a smaller average-case time complexity.

Aho-Corasic (AC) [27], Commentz-Walter [28] and Wu-Manber [29] are typical multi-pattern matching algorithms, which search for a set of patterns each time. Aho-Corasic [27] is the most well-known multi-pattern matching algorithm, which combines the characteristics of KMP and finite-state machine. The Commentz-Walter [28] algorithm is an extension of Boyer-Moore algorithm to support fast multi-pattern matching. Based on them, there are also some methods proposed for time or space efficiency. As an example, AC-BM algorithm [30] combines Aho-Corasic and Boyer-Moore algorithms and is used in Snort [9]. Tuck et al. in [31] optimize Aho-Corasic algorithm using a bit-map for a compressed representation of state nodes and path compressions in an AC tree. Wu-Manber [29] and its variation in [32] belong to multi-pattern and multi-stride algorithms, which process more than one characters each step. Multi-stride algorithms require a sequential comparison of patterns and their stride largely relies on pattern characteristics, and these properties restrict their usage.

Optimizations on fixed pattern algorithms also include Bit-split state machines in [33], Trie Bitmap Content Analyzer in [34] and parallel Bloom filter based matching in [35] etc.

2.2.3 Regular Expressions Matching Algorithms

With the development of NIDS, fixed pattern matching based DPI cannot keep up with the fast growth of pattern rule sets. In recent years, the rich expressiveness of regular expressions makes it dominant in NIDS. NFA and DFA are two classical and equivalent representations of regular expressions.

- **NFA algorithms**

An NFA can be constructed in O(n) time and takes O(n) memory for a regular expression of length n. NFA processes a single text character in O(n) time, thus the time required to search through text of length m is O(mn) and the required memory is O(n). As an improvement to the original NFA approaches, Sidhu et al. in [36] reduce the processing time to O(1) (one clock cycle per character), and the new algorithm requires O(n^2) memory, which is an affordable tradeoff.

The main flaw of NFA is that it can have multiple active states at the same time. This drawback can be easily improved using hardware, where multiple automatons for different regular expression groups can be simultaneously active in parallel. An NFA can be decomposed into multiple modules running in parallel processing engines; for faster speed, each engine has only one active state at any time.

- **DFA algorithms**

DFA can be constructed in O(2^n) time and the construction of DFA takes O(2^n) memory size, where n is the length of regular expressions. DFA can process a character in O(1) time, so that the time required to search through a text of length m is O(m) after the DFA is constructed, and the required memory is O(2^n). Most text searching algorithms are based on DFA and use optimization techniques to reduce its construction time and memory requirements.

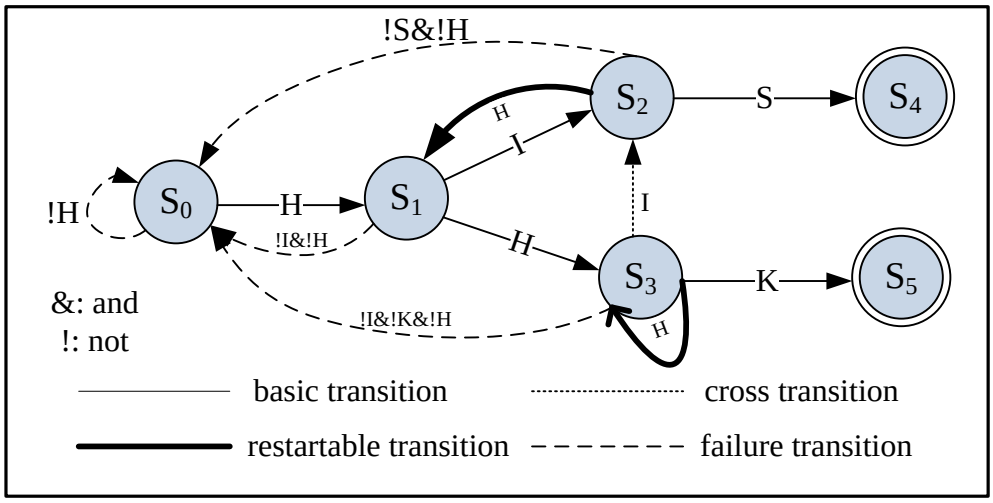


Figure 2-4 A DFA state machine for {HIS, HHK}

Figure 2-4 shows an example of DFA for patterns {*HIS*, *HHK*} constructed with Aho-Corasic algorithm. There are four types of transitions: 1) basic transition moving along each pattern, such as the lines from S_0 to S_1, S_1 to S_2 and S_2 to S_4; 2) cross transition that transits from one pattern (e.g. *HIS*) to another pattern (e.g. *HHK*), such as the dot line from S_3 to S_2; 3) restartable transition that moves from current state to the next states of the beginning state S_0, such as the bold edges from S_2 and S_3 to S_1; and 4) failure transition that moves from current state to the beginning state S_0, such as the dashed lines from S_1, S_2 and S_3 to S_0. Figure 2-4 is a simple illustration of DFA transitions; for more complex combination of patterns, cross transitions possibly outnumber all the other kinds of transitions.

This thesis works on DFA based algorithms. Pattern writing, pattern groupings, pattern splits and multiple stride DFAs are popular optimization methods proposed to reduce DFA memory consumption or improve its matching speed.

1) Pattern rewriting

Pattern rewriting explores the inherent characteristics in a single regular expression. Some types of regular expression cause an exponential growth in the number of DFA states. As a solution, Yu et al. in [17] propose pattern rewriting for two types of regular expressions: 1) patterns starting with '^' as a fixed prefix and

with a length restriction *j*; 2) patterns with a length restriction *j* where a wildcard or a class of characters overlaps with the prefix. These patterns inevitably produce state blowups, but the replaced patterns reduce exponential or quadratic size of states to a linear number of states, which dramatically reduce DFA sizes. But the drawback of rewriting pattern rules is that it is effective for only a small number of regular expressions in NIDS.

2) Pattern grouping

Pattern grouping works on the combination of regular expressions. The state explosion in DFA generation for a large number of regular expressions can be prevented by partitioning them into groups and generating multiple DFAs for these groups. During pattern grouping, patterns are added heuristically to construct a DFA within available memory size. The essential criterion is to make sure that there are as few interactions as possible among regular expressions within one group, and DFAs within different groups do not interact with each other. For example, given two regular expressions RE_1 and RE_2, in consistent with DFA_1 and DFA_2, the combined DFA_{12} is created. If the size of DFA_{12} does not exceed the sum of the sizes of DFA_1 and DFA_2, the two patterns can be clustered together. For multiple DFAs, a single processor runs each of them sequentially, while multi-processor can run several DFAs in parallel. But the memory reduction is at the cost of the increase in memory bandwidth. The running of *k* DFAs in parallel means *k* concurrent state traversals for each character.

3) Pattern splits

Pattern splits or DFA splits target at high speed by splitting apart state machines. The bit-split string-matching engine, proposed by Lin et al. in [33], uses multiple parallel DFAs for the split input characters. This method extracts the 8 bits ASCII character of the basic state machine, into 8 binary state machines in parallel, whose alphabet only contains 0 and 1. Each tiny state machine searches for a portion of the rules and a portion of the bits of each rule.

Alternatively, instead of treating the whole pattern equally, Kumar et al. in [37] separate patterns into smaller parts. State automata are differentiated as fast path and slow path. The frequently visited states are stored in a fast path automaton, and the rest of less frequently accessed states are stored in a slow path automaton. The fast automaton is always active while the slow automaton is activated only if the network packets match the fast automaton. In normal condition, a very small fraction of network packets is matched by the fast path and continues to be inspected by the slow path. Therefore, active automaton keeps fast and compact.

However, this kind of division is vulnerable to Denial of Service (DoS) attacks, which repeatedly send specially constructed data that matches the fast path. In this case, the slow path becomes the bottleneck and system throughput will be seriously affected. A practical solution to this problem is to use a history table keeping the accessing times of slow path with the incoming traffic [37].

4) Multi-stride DFAs

Multi-stride DFAs further accelerate processing throughput in the context of small DFAs. Multi-stride DFA increases the performance of pattern matching by scanning multiple characters of a packet at the same time, instead of byte-by-byte inspection. Hua et al. in [38] propose a variable stride DFA, VS-DFA, which partitions patterns into variable size blocks using a fingerprint scheme. Another software level algorithm, multiple-stride DFA (MS-DFA) in [39], is based on similar block-based DFA concepts. MS-DFA groups states or transitions into several coarse-grained and variable-size blocks, where each block uses specific methods to optimize storage requirements and performance. These blocks can be uniquely identified through both the pattern and the input stream, so that the multiplied memory costs as in previous approaches like [40] can be avoided.

When it requires even higher throughput, the multi-stride algorithms can be implemented on hardware platforms. Sugawara et al. in [41] process multiple input characters each time using a Suffix Based Traversing (SBT) of a modified AC.

Through a specially designed hardware approach, Brodie et al. in [42] allow simultaneous traversal of multiple transitions to increase throughput. Based on FPGA, Lu et al. in [40] propose a multi-character architecture consisting of multiple parallel DFAs, called transition-distributed parallel DFAs (TDP-DFA), which can achieve 40Gbps wire speed processing. However, it requires very large memory since multiple identical DFAs need to be stored.

Currently, the idea of multi-stride is not widely used in industry because it relies on specific hardware for parallel comparison. Besides, multi-stride requires huge memory and the pattern expansion from multiple input characters brings larger chances of state explosion.

2.3 DFA Memory Compression Techniques

Given the increasing importance of DPI, it is imperative to implement the low-memory and high-throughput pattern matching. Fast speed can be achieved by using DFA on hardware platforms, but DFA algorithms are restricted by the limited memory resources on hardware platforms.

Besides, the NFA-to-DFA transformation possibly causes state explosion. RegEx [43] etc. tools can construct an NFA directly from regular expressions, and convert the NFA to a corresponding DFA. Theoretically, given an *n*-state NFA, the maximum number of its DFA states is 2^n [20]. More often, when multiple regular expressions are compiled to a single DFA, the combination of different regular expressions leads to state blow-up.

The state explosion of DFA can be solved by alphabet reduction, transition reduction, state reduction and novel types of state machine techniques.

2.3.1 Transition Compression

It is observed in state automaton that a large number of states especially adjacent states have similar transitions with the same input characters.

- **D²FA**

Kumar et al. in [16] introduce a delayed input DFA - D²FA to reduce memory of redundant transitions. In terms of the DFA built from real datasets, each state has more than 50 distinct transitions on average. If two states have identical outward transitions, D²FA makes one state point to the other state through a default transition.

The problem of reducing maximum memory equals to constructing a maximum weight spanning tree in an undirected graph, where a node is a state and a path is a transition or a default transition of a DFA. The weight of a path is associated with the number of input symbols on its related default transition. Nevertheless, the maximum weight spanning tree results in a larger bandwidth overhead due to long default paths, which can be constrained by a diameter bound for the number of edges in the longest path. The D²FA construction requires $O(n^2 logn)$ time complexity and $O(n^2)$ space complexity, where n is the number of states in DFA [16].

Figure 2-5 is an example of D²FA, where the bold edges are called default transitions. The matching of D²FA takes the default transitions, whenever a labeled transition is missing for an input. It can be seen from this example that D²FA is compact but requires multiple memory accesses, thus it is not desirable in off-chip architecture.

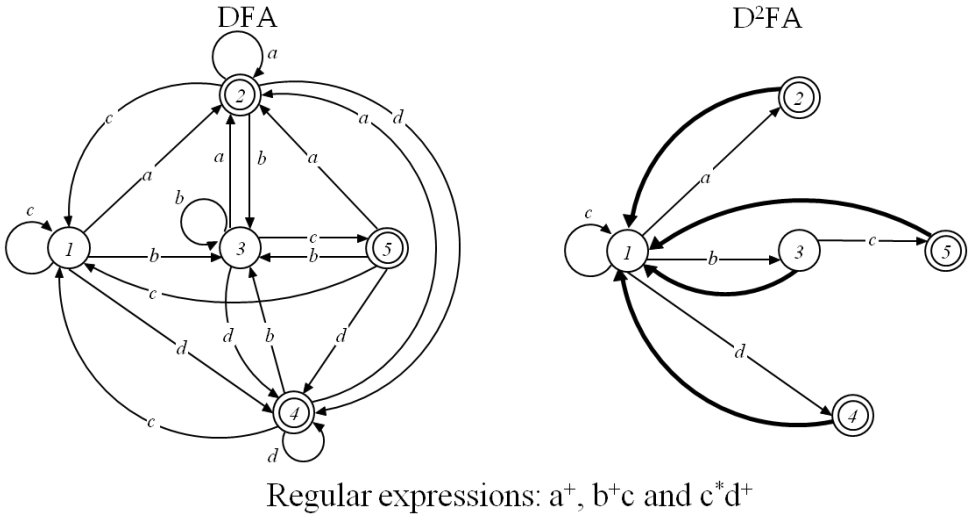


Figure 2-5 An example of DFA and D²FA

- **CD²FA**

Kumar et al. further propose *Content Addressed Delayed Input DFA* (CD^2FA) in [19] to guarantee one state traversal per input character. The main defect of D^2FA is its unnecessary memory accesses. States in CD^2FA have content labels to indicate associated characters and their follow-up default states, so that it can predict further steps for an input character and choose an optimal one. Figure 2-6 shows an example of three states S_0, S_1, S_2 in CD^2FA and bold edges denote default transitions. The content labels denote the characters that can be accepted at each state. State S_0 is the root of the three states, state S_1 has labeled transition for *c* and *d* in addition to its default transitions for other characters to state S_0 and state S_2 has labeled transition for *a* and *b* in addition to its default transitions for other characters to S_1. Assume the current state is S_2, with an input character *a*, it goes to the next state from S_2; with an input character *c*, it goes to the next state from S_1; with an input character *e*, it goes directly to S_0 as next state. In this way, the content addressing indicates next state and assists the reduction of the number of transitions per each input character.

CD^2FA can maintain the throughput of uncompressed DFA while it uses 10% of the space required by a conventional compressed D^2FA with only specified transitions [19]. In experimental evalution of [19], the author also compares the performance using the data cache, CD^2FA achieves two times higher throughput as compared to an uncompressed DFA. Because of the much smaller memory footprint, CD^2FA has higher cache hit rates (>60%) and hence the throughput is also increased. But this result is based on the use of data cache, and the input data stream results in a very high matching rate.

Although CD^2FA shows excellent results in memory and speed, its creation and maintenance costs are more expensive. The memory compression of DFA in Chapter 3 is based on D^2FA; the use of labels on transitions of CD^2FA complicates the analysis of memory contents. In order to achieve a faster throughput, the work in Chapter 3 introduces new states.

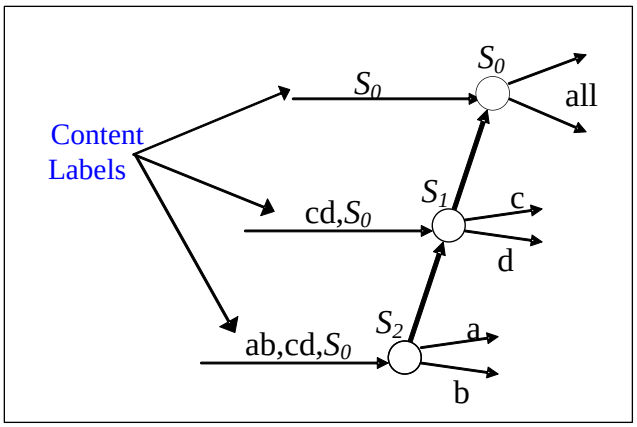


Figure 2-6 Default transition optimization with content labels in [19]

2.3.2 Alphabet Compression

Alphabet compression is another technique to reduce memory requirement by a different encoding scheme to represent compressed DFAs. The basic idea is that multiple characters can fall into one group as a super-character if they label the same transitions everywhere in the automaton. Alphabet compression is orthogonal to other optimization techniques and can be employed together. However, this method is not effective for a large number of regular expressions, because it is expensive to run with many super-characters.

- **A-DFA**

The Alphabet-DFA (A-DFA) in [44] uses alphabet reduction with stride doubling, a technique to reduce the memory bandwidth requirement. It first clusters the characters with similar behaviors and the stride doubling is then applied to transform the DFA representation. Kong et al. in [45] use multiple alphabet translation tables to obtain an even smaller set of alphabets. Based on the new translated alphabets, a heuristic algorithm clusters states in DFA. But this method has overhead of multiple translation tables during the inspection process.

2.3.3 Solutions to State Explosion Problems

The fatal drawback of DFA is the state explosion, which is mainly caused by the

usages of dot-star terms, counting constraints and back-references in regular expressions, the solutions of which are explained as follows.

- **Solution for dot-star terms**

A primary use of dot-star terms is to detect occurrences of sub-patterns separated by an arbitrary number of characters. In most cases, dot-star terms do not cause state blow-up when individual regular expressions are compiled in isolation. However, dot-star terms add complexity when distinct regular expressions are compiled together. For example, the composite DFA for regular expressions "*ab.*cd*" and "*ef.*gh*" requires $O(n^2)$ memory, where n is the number of characters. The non-deterministic sub-patterns can be transformed to deterministic form using pattern rewriting techniques described in [17]. Hybrid-FA in [21] also deals with dot-star problem and will be introduced in Section 2.3.4.

- **Solution for counting constraints**

Counting constraints are usually used to detect buffer overflow situations in Snort [9]. But the counting constraint produces unstable memory and bandwidth requirements. Moreover, the condition gets dramatically worse when multiple regular expressions with counters are compiled together into a combined DFA. It requires a large number of states in the DFA corresponding to counter *n*. Becchi et al. in [46] propose an extension, to either DFA or NFA, with a counter added to transition condition, so that the counting constraint can be expressed by two states connected through a labeled transition for counters. This however slows down the matching speed since the state machine needs to refer to counters in memory during transitions with counting constraints.

- **Solution for back-references**

Back-reference means that a matched substring in the prefix, which is enclosed within capturing parentheses, should be matched again later in the input string. For example, an expression "*(abc|bcd).\1y*" means that after reading "*abc*" or "*bcd*", and an arbitrary symbol, the subsequent string should be exactly as the one in the

parentheses, "*abc*" or "*bcd*", depending on which one was matched previously. Its difficulty is the uncertainty of the second appearance of the sub-pattern in parentheses. An intuitive solution is to keep the matched parts of the input text. But it conflicts with intrinsically memory-less property of DFA that it does not remember previously processed sub-strings, thus back-references cannot be directly supported through finite automata. Extended-NFA in [46] uses a label, which indicates whether the previous substring within parentheses has appeared again, to handle back-references with an NFA-like operation. This scheme is at the cost of conditional transitions, which must be represented in a compact way.

2.3.4 Novel Types of DFA

This section introduces new types of DFAs, including Lazy-DFA [47], History-based-FAs [38], XFAs [22][48], δFA [20], Hybrid-FA [21], CDFA [38] etc. algorithms that modify the structure of DFA for the benefits of memory or speed.

- **Lazy-DFA**

Bro system [10] uses Lazy-DFA [47] to reduce the memory consumption of conventional DFA. Lazy-DFA keeps a subset of the DFA that matches the most common strings in memory; for uncommon strings, it extends the subset from the corresponding NFA at runtime. However, malicious senders can easily construct packets with uncommon strings to keep the system busy and slow down the matching process. As a result, the system will start dropping packets and malicious packets can sneak through.

- **History-based-FA**

Kumar et al. propose history-based-FAs in [38], which consists of a finite state machine coupled with a history data structure. It is known that a DFA contains a state for each possible combination of NFA-states, which can be concurrently active. Some of these combinations are similar and can be differentiated with additional history information. This property can be utilized to reduce the total number of states and

transitions, by storing some matching history in a fast and small cache. But this approach also has some drawbacks. Firstly, the number of NFA-states stored in the history can affect the number of transitions as well as the complexity of the conditions to be evaluated. Second, history-based-FA is intended to be processed by a single thread at the cost of more complex processing on average case.

- **XFA**

Another novel structure is XFA proposed by Smith et al. in [22][48]. Similarly to history-based-FAs, the basic idea is to extend classical DFA through external variables and instructions, so as to avoid duplication of states. Evaluated with a larger number of NIDS signatures, XFA shows similar time complexity as DFAs and similar space complexity as NFAs. However, the disadvantage is that it only addresses counting constraints located at the end of regular expressions.

- **δFA**

Ficara et al. in [20] present a compact representation scheme, Delta Finite Automata (δFA), based on the observation that most adjacent states share a large part of identical transitions. Compared with D^2FA [16], δFA stores only the differences between adjacent states (i.e. "parent-child" states), and restricts the number of traversal times per character. However, the difference between the current and the next state must be computed on each state traversal. Thus it needs $O(|\Sigma|)$ time, where Σ is the number of characters, to update transitions of current state as a new state is reached.

- **CDFA**

Song et al. in [49] propose a Cached Deterministic Finite Automate (CDFA) model to implement a modified Aho-Corasic (AC) algorithm, called AC-CDFA (ACC). Suppose the root state of DFAs is at level one, it is observed that a large fraction of transitions on AC-DFA are backward to states at the first three levels. It extends DFA by allocating some registers as cache in its model. Compared to basic DFA structure in Figure 2-7(a), where next state is obtained by referring to memory

and input character, CDFA in Figure 2-7(b) adds a cache in its model, where the next state is determined by input character, current state and *N* cached states in cache (*N*=1 in [49]). Apart from the transition function, CDFA also requires a cache function. Although the authors of this work mainly focus on simple strings patterns, they prove that CDFA works equally for multiple regular expression matching.

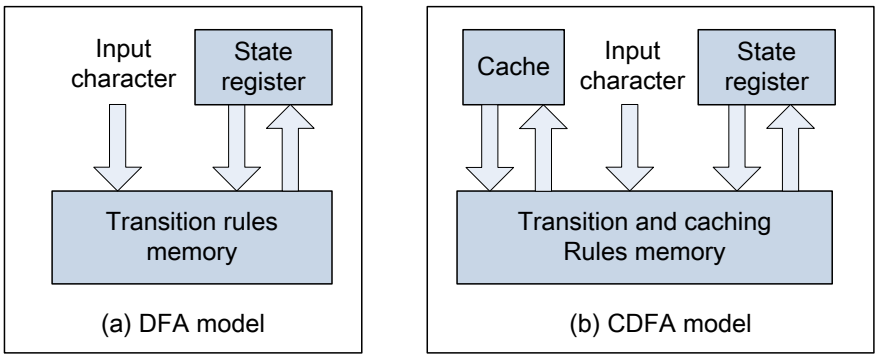


Figure 2-7 DFA and CDFA model

- **Hybrid-FA**

Hybrid-FA, proposed by Becchi et al. in [21], is a highly parallelizable data structure. It is designed to deal with two difficulties in constructing DFAs, the "dot-star" terms and the "counting constraints". Hybrid-FA is composed of head-DFAs and tail-NFAs, each of which handle different sub-patterns and can be processed by separate execution threads. It is assumed that the processing of non-malicious traffic happens within a fast head-DFA. Hybrid-FA uses partial subset construction to avoid the expansion of all combinations of special states. Hybrid-FA is a viable alternative method when a DFA cannot be feasibly built on existing hardware resources.

As an example in Figure 2-8, Hybrid-FA is composed of a head-DFA and many tail-NFAs acting as a compromise between a pure DFA and a pure NFA solution. The tail-DFA is exemplified from tail-NFA, depending on the input string. When a transition is activated by an input character, the corresponding tail-NFA is converted to a tail-DFA, while other tail-NFAs stay unchanged.

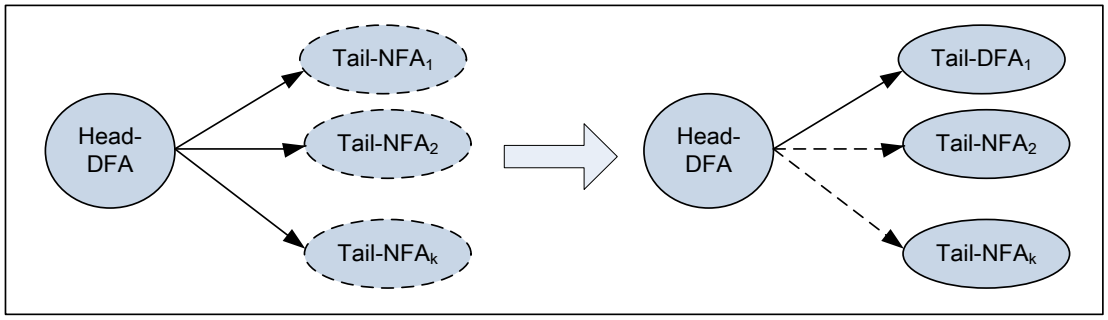


Figure 2-8 Hybrid-FA exemplifications from tail-NFA to tail-DFA.

In Hybrid-FA, the dot-star terms only create linear increases in the number of DFA states, since a tail-DFA will be replicated once for every occurrence of a dot-star term in other regular expressions. When transforming a tail-NFA into a tail-DFA, the construction operation is interrupted at those NFA states whose expansion would cause state explosion and keep it as tail-NFA. The structure of a Hybrid-FA is specifically composed as follows: the starting state will be a DFA-state; the NFA part of the automaton will remain inactive till a border-state is reached; and there will be no backwards activation of the DFA coming from the NFA.

2.4 Bloom filters

Bloom filters are widely used in data based applications and network processing. Proposed by Burton Bloom in [50], Bloom filter is a fast and memory efficient randomized data structure with a group of hashing tables and functions for membership queries. Hash tables provide excellent performance for data representation and hash functions have linear time complexity for search, insertion and deletion operations. The simplicity of Bloom filter also makes it suitable for hardware implementation.

Bloom filter is another popular approach for DPI. Considering its application in DPI, the patterns to be matched are mapped through a number of hash functions associated with Bloom filters, and generate an array with zeros and ones. The payload bytes are also mapped to array positions, it reports a match only if all the mapped

entries equal to one; or else, it reports a non-match. But the results also include false positives and need to be further analyzed if exact matching is required.

Optimizations and variations of Bloom filters are proposed to meet specific requirements in their applications. Based on original Bloom filter structure, four types of modification methods on hash tables or hash functions are explained, each of which helps to achieve a better memory or speed efficiency.

2.4.1 Overview of Bloom Filters

In recent years, Bloom filter receives extensive attention in network applications as a solution to the insufficiency of bandwidth, the storage of large data set and network security. For example as a distributed caching scheme, the Summary Cache in [51] uses Bloom filters to compactly represent URLs stored in a cache. Dharmapurikar et al. in [35] use multiple engines of Bloom filters in parallel on FPGA to speed up DPI in NIDS, where security patterns are densely mapped to entries in the much smaller hash tables. Bloom filters are also used in [52] to efficiently save the log of all packets received in the last hour after an attack. Kumar et al. in [53] use Space-code Bloom filter to support traffic monitoring in high speed networks.

The use of Bloom filters includes training stage and searching stage. Suppose a Bloom Filter has k hash functions $h_i(x)$, i=1…k. In the training stage, each of n items in set $S = \{x_1, x_2 ... x_n\}$ is mapped to k positions in the hash table over the range of {1, …, m} (m>n). Figure 2-9 shows the mapping of item x to the hash table. The values in the k hashed positions are set to 1. In the searching stage, it produces a non-match if at least one of the k mapped positions is 0, and produces a match if all the positions are 1.

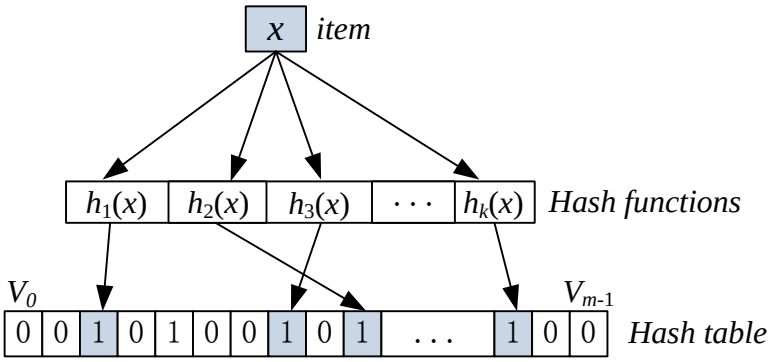


Figure 2-9 Bloom filter structure

The match results reported by Bloom filters include false positives, which mean an item is actually not a member of the set but the Bloom filter returns yes. The false positives of Bloom filters require extra verification time for further membership confirmation. Equation 2-1 defines the false positive rate f [12], where $p = e^{-nk/m}$.

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k = \left(1 - p\right)^k \tag{2-1}$$

Given m and n, the optimal k can be calculated by Equation 2-2.

$$\begin{aligned} f &= \exp\left(k \ln\left(1 - e^{-kn/m}\right)\right) \\ g &= k \ln\left(1 - e^{-kn/m}\right) \\ \frac{dg}{dk} &= \ln\left(1 - e^{-kn/m}\right) + \frac{kn}{m} \frac{e^{-kn/m}}{1 - e^{-kn/m}} \end{aligned} \tag{2-2}$$

The false positive f is minimized in the conditions of Equation 2-3.

$$\begin{aligned} &\frac{dg}{dk} = 0 \Rightarrow k_{\min} = (\ln 2)\left(\frac{m}{n}\right) \\ &f_{\min} = f(k_{\min}) = (1/2)^k = (0.6185)^{m/n} \end{aligned} \tag{2-3}$$

There are a number of variations based on the basic Bloom filters, including Counting Bloom Filters (CBF) [51], *d*-left CBF [54], Multi-Layer Compressed CBF [55] and Dynamic Count Filters (DCFs) [56], all of which are suitable for hardware implementations and have been used as candidates for DPI. Other variations include Compressed BFs [57], Space-code BFs [53], Spectral Bloom Filters [58], Dynamic Bloom Filters (DBF) [59][60] etc. algorithms, each of which aims at reducing

memory consumption or decreasing false positive rate. Next three typical novel Bloom filter structures are presented: CBF [51], SBF [58] and DBF [59].

- **CBF**

Fan et al. introduce the idea of Counting Bloom Filter (CBF) in [51] to support the insertion and deletion of an item with a counter, instead of a single bit, at each entry of the hash table. Basic Bloom filters cannot support online update of set members. A new item is inserted by setting all the mapped bits to 1. On the other hand, deleting an existing item is not feasible since the mapped bits of this item are possibly associated with other items in the dataset. The use of counters supports the searching, insertion and deletion of items. Searching an item is to check if all the mapped entries are non-zero. Adding or deleting an item is implemented by increasing or decreasing all the mapped counters by one. But CBF has the possibility of counter overflow.

- **SBF**

Spectral Bloom filter (SBF) in [58] extends CBF to multi-sets with information of appearing frequencies referred to as spectrum. The spectral expansion allows the filtering of elements whose number of multiplicities is below a threshold given at query time. It is designed for data stream applications or tracking large flows in network traffic. Bloom filter or CBF approaches are not adequate when dealing with frequencies of multi-sets where items may appear hundreds of times. SBF consists of a compact base array of a sequence of counters. Each counter in SBF structure dynamically varies its size so that it has the minimum necessary bits needed to counter the number of items hashed to the entry. While the counter space in SBF is kept close to its optimal value, SBF requires complex index structures in order to support the flexibility of having counter with different sizes.

- **DBF**

Dynamic Bloom Filters (DBF), introduced by Guo et al. in [59][60], allocates memory in an incremental way to support a concise representation and approximate membership queries of dynamic sets. It is suitable in a scalable environment, where

the final size of a dataset is not known in advance.

In summary, Table 2-1 compares the characteristics of three different Bloom Filters, each of which is suitable for particular applications.

Table 2-1 Comparison of CBF, SBF and DBF

Functions	Counter	Cache	Access	Update Cost	Counter overflow
CBF	Static	Smaller	Fast	None	Yes
SBF	Dynamic	Larger	Slow	High	Eventually
DBF	Dynamic	Larger	Fast	Low	No

2.4.2 Optimizations of Bloom filters

The enormous number of patterns in DPI requires more efficient use of hash tables in Bloom filters. Besides, when CBF is used to support the update of security rules, it has the counter overflow problem, which cannot be tolerated in network applications. With modifications on hash tables or hash functions, the original Bloom filter structure can be improved by the following four practical methods: 1) hash table with multiple segments; 2) compressed hash table with multi-layers; 3) hash table with dynamic sizes; and 4) d-left hashing.

1) Hash table with multiple segments

For the purpose of a more balanced distribution, the hash table in Bloom filter can be divided into multiple segments, instead of a single hash table, assisted by segmented hashing functions. Segmented hashing partitions the hash table into *N* segments, among which an item is inserted in the most sparse segment in order to balance data distribution [61]. By allocating the elements as evenly as possible among hash table memory, it reduces the false positive probability for searching an element.

The fundamental idea of Bloom filter with segmented hashing is the selective filter insertion, which keeps the whole table balanced by minimizing the number of non-zero counters distributed in all the segments of a Bloom filter. Considering a Bloom filter with *N* segments, the insertion of a new item is mapped to *N* entries and is only inserted in the one with the smallest counter. It probes every segment and

inserts the new item into the first vacant place.

For example in Figure 2-10, a new element k_i is mapped to the second position, and after checking the second entry of each segment, k_i is stored at the last bucket and the counter is updated from empty to 1. Next another new element k_{i+1} is mapped to the fourth position, and there is no vacant place at the fourth entry of each segment, k_i is stored at the first bucket with a minimum counter value of 1.

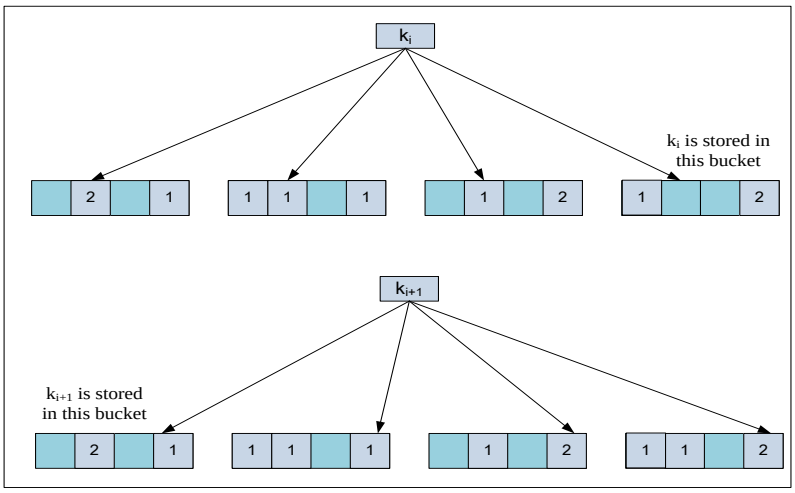


Figure 2-10 Data structure for Bloom filter with multiple segments

The false positive rate of segmented Bloom filter is analyzed as follows. Suppose there are N independent Bloom filters, each of which corresponds to one segment. And there are now N opportunities for a false positive. Equation 2-4 defines the probability P_i of having i false positives, where f is the false positive rate for a single Bloom filter.

$$P_i = \binom{N}{i} \times f^i \times (1-f)^{N-i} \tag{2-4}$$

The total false positive rate P shown in Equation 2-5 is the sum of all possible P_i.

$$P = \sum P_i = \sum_{i=1}^{N} \binom{N}{i} \times f^i \times (1-f)^{N-i} \tag{2-5}$$

Since multi-segment Bloom filter reduces the collision chances, it has a smaller memory size, but needs to search at most N places. If each segment is evenly loaded, $N/2$ segments, on average, will need to be probed before the key is found. The search

can be performed in parallel at the cost of *N*-fold increase in memory bandwidth.

2) Compressed hash table with multi-layers

For memory efficiency, hash table in Bloom filter can be compressed by using multi-layers. Network applications, with a huge mass of data such as P2P or web cache sharing, require the most space efficient data structure. Multi-layer CBF (ML-CBF) in [55] further reduces memory consumption for network applications. It utilizes a stack of Hash-Based Vectors (HBVs) in addition to basic CBF vector (CBFV) and resorts to multiple levels to solve counter overflow. If the counter on one level saturates, it will be hashed to a higher level. In order to find the current counter, if the counter of the hashed position saturates, it will hash with the previous hashed value using a higher level hash function. This process continues till it finds a normal value or to the uppermost level. Results in [55] show that this structure can save memory and solve counter overflow.

Figure 2-11 shows an example of Hash-Based Vectors for a ML-CBF. There are 3 bits on basic level CBFV, 2 bits on the first level HBV_1 and 1 bit on the second level HBV_2. The address starts from 0. Considering the insertion of an element *s*, since $h_i(s)$ =4 and the value at position 4 is saturated (CBFV[4]=7), it goes to the first level; but the mapped entry is full again as $h_{k+1}(4)$=6 and $HBV_1[6]$=3, the insertion of *s* proceeds to the second level, where $h_{k+2}(6)$=1 and $HBV_2[6]$=1.

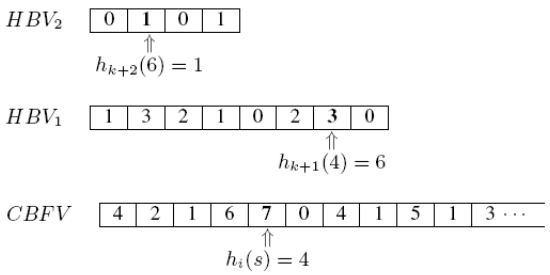


Figure 2-11 Hash based structure of ML-CBF [55]

3) Hash table with dynamic sizes

As an effective solution to counter overflows of CBF with reasonable memory

consumption, hash table in Bloom filters can be designed with dynamic sizes. Dynamic Counting Bloom Filters (DCBF), proposed in [56], is a dynamic and space efficient representation of CBF. It is observed in CBF that the decision of having the same size of all these counters entails the fact that many bits in counters are not used. Therefore, CBF scheme can be improved in terms of memory consumption. Instead of fixed bit width for each entry, DCBF also uses the dynamic allocation of bits depending on the distribution of entry loads.

DCBF maintains two counting vectors. The first one is a basic CBF (CBFV) with counters of fixed size, while the second one is the Overflow Counter Vector (OFV) to keep track of the number of overflows for each element in the first vector. The size of the counters in OFV dynamically changes to avoid saturation; however, it also implies that each update requires a structure rebuilding. For the DCBF structure in Figure 2-12, and the counters of entries 3, 5 and *m* overflow, where their real values are values in CBFV plus values in OFV multiplied 256. Take Entry 3 as an example, its counter value is 256 plus 1, i.e. 257.

Compared with SBF, which requires an index structure to support counters with different sizes, DCBF trades counter memory space for fast access, since the fixed-sized counters allow fast read and write. Besides, the cost of rebuilding DCBF is much smaller than that of SBF.

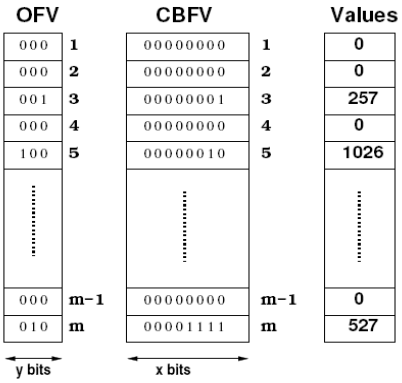


Figure 2-12 Dynamic counting filter structure

4) D-left hashing function

D-left hashing is a commonly used method in Bloom filter, where a hash table consists of *n* buckets. Initially the *n* buckets are divided into *d* disjoint sub-tables, consisting of *n*/*d* buckets. An element is hashed to a collection of *d* possible buckets, each of which belongs to a separate sub-table, and the element is placed in the bucket that contains the smallest number of elements.

The structure of *d*-left hashing is explained as follows. Firstly, a *d*-left hash table consists of *d* sub-tables, each of which has a number of buckets. Second, each bucket has many cells and a counter, indicating the number of used cells. Third, each cell, with a fixed number of bits, stores part of the hashed fingerprint. Since each bucket has a fixed number of cells, in order to avoid bucket overflow, the load should be distributed as evenly as possible.

Take an example in Figure 2-13, a hash function, $H : U \rightarrow [B]^d \times [R]$ maps a new key *x* to a fingerprint. The fingerprint consists of two fields: first *d* sections are used as the address thus there are *d* choices for each input item; and among *d* choices, the remainder is stored in the least loaded bucket. In this example, the *d* addresses of *x*'s fingerprint point to *d* buckets respectively, and the remainder of *x*'s fingerprint is stored in a cell of the bucket of sub-table 2, with the smallest counter of "0", which is updated to 1 after insertion. If there are multiple empty buckets, the new element is inserted to the one in the first appeared sub-table. The lookup of an element has to search all the *d* buckets simultaneously. Compared to normal hash table, the elements in above structure are more evenly distributed, thus the chance of hash collision is greatly decreased.

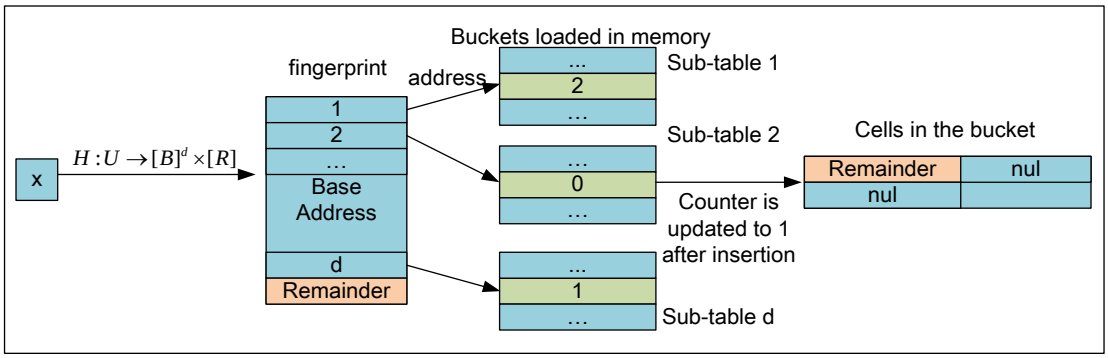


Figure 2-13 D-left hashing structure

While insertion and lookup of an element is more easily achievable, a problem exists for the deletion of an element. It has no way to indicate whether two elements are mapped to the same bucket or not. Thus the corresponding remainder might be found in more than one of the *d* choices, since it is possible that another element may have generated the same remainder and placed it in another of these *d* buckets. In this case, d-left hashing cannot differentiate remainders and cannot decide which copy to delete.

As a variation of CBF based on *d*-left hashing, d-left CBFs (dl-CBF) stores fingerprints that use less memory and avails to dynamically insert or delete a set of items [54]. False positive possibilities can be eliminated by comparing with the stored fingerprints. When memory is a fixed value, the probability of bucket overflow is reduced as parallel choice *d* gets larger.

Figure 2-14 is an example of dl-CBF. D-left hashing generates *d* addresses in sub-tables and a fingerprint. To balance the distribution of *m* elements, the hash table is divided into *d* sub-tables, each of which has 3 buckets, and each bucket contains 8 cells. A cell can store 4 copies fingerprints for the elements at most, and needs 2 bits for the counter. The advantage of dl-CBF is its simplicity in constructing and maintaining data structures, despite the fact that dl-CBFs also have the limitation of potential counters overflow, and the need for an additional fingerprint for each cell in its data structure.

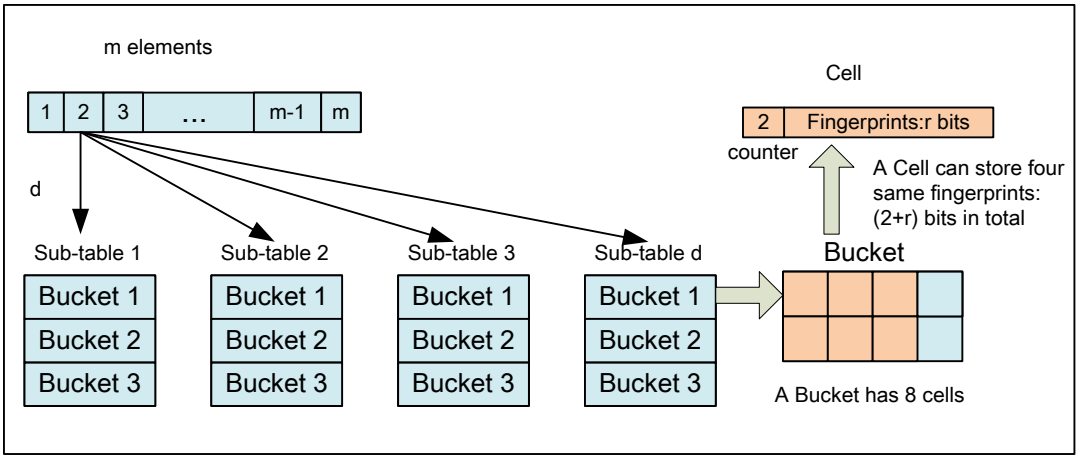


Figure 2-14 An example of dl-CBF

2.4.3 Bloom filter on Hardware Platform

Bloom filter is often used on hardware platforms like FPGA. Bloom filter based FPGA schemes divide its memory into on-chip SRAM and off-chip DRAM. By efficiently allocating tables on different kinds of memories, it could improve processing speed on basis of existing hardware resources. Rather than storing the whole table on-chip, tables in Bloom filters can be separated into on-chip and off-chip tables depending on their data access rates. Probabilistic on-chip filters are used to avoid off-chip memory accesses and enable deterministic hashing performance.

For memory efficiency, the hash tables of Bloom filter can be placed on different memory levels. For example, the lookup tables for a basic Bloom filter or CBF are kept on chip, and the data used for further analysis of false positive cases can be stored off-chip and will be accessed after matching the lookup table of Bloom filter at the first level. For multiple-level hashing tables and its related Bloom filter, the memory can be allocated more flexibly. One of the Bloom filter algorithms that achieve remarkable on-chip memory reductions is the Peacock hashing, proposed by Kumar et al. in [62]. By storing the largest table off-chip, Peacock hashing achieves 90% reduction compared to the original Bloom filter size [62], which makes it an attractive choice for the implementation of a hash accelerator on a network processor.

2.5 Hardware Approaches

Due to the complexity of rule sets and the need to scan the whole payload, hardware-based accelerators with high speed parallel processing engines are utilized for high-speed pattern matching. On the other hand, DPI also requires low power consumption as well as easier configuration for system maintenance and update.

Generally, there are three kinds of hardware implementations including FPGA, TCAM and multi-core NP. A major limitation of FPGA and other hardware based approaches is that they cannot swiftly support online update of rules, so that the security rules are required to be modified offline. This is due to the fact that it takes considerable time to re-synthesize the design and re-program FPGA core etc. hardware platforms. In particular, each approach has its advantages and drawbacks.

In terms of comparison on power consumption, FPGAs impose a larger power overhead compared to custom silicon alternatives. In view of power consumption, typically, cell-based ASICs offer the lowest power consumption. They are followed by structured ASICs and then FPGAs. The power overhead limits integrations of FPGAs into portable devices.

Next each of them will be briefly introduced.

2.5.1 FPGA Solutions

FPGA can run fast with high frequencies and is very efficient as it can be customized for the desired functions. It also supports large degree of parallelism, therefore, DPI system can instantiate multiple engines running in parallel for higher speed. Compared to Application-specific Integrated Circuit (ASIC), programming of either fixed string or regular expressions on FPGA is much easier; and new configuration, such as pattern set update, is performed by downloading new programming code. Patterns and tables could be stored on on-chip memory or off-chip DRAM etc. In recent years, a number of high-speed pattern matching

solutions are implemented on FPGAs, including DFA based methods [41], NFA based methods [36], comparator based methods [36] and Bloom Filter based methods [35] etc.

2.5.2 TCAM Solutions

TCAM is also an attractive candidate for fast pattern matching. For example, F. Yu et al. used TCAM for fast pattern matching in [63]. The advantage of TCAM is the parallel searching of tables. But it has some disadvantages as well. Firstly, TCAM only reports one entry with the lowest index even if the input string matches multiple TCAM entries. In practice, a priority encoder is always added to make the one with the longest prefix match appearing in front. Second, it has the highest power consumption among different hardware platforms. Third, it is hard for TCAM bits to represent some special symbols in regular expressions, such as the negation or range of length. These drawbacks limit its application in network packet processing.

2.5.3 Multi-core NP Solutions

Multi-core NP platform is more flexible and has higher computational capabilities. The commercially used Network Processor Units (NPU), such as Intel IXP 1200, 2400, 2800, contains 8 to 16 cores. These platforms have been widely used in academia and industry over the last ten years. Software NIDS approaches like Snort or L7-filter can be implemented in NPU.

The multi-core NP architecture contains multiple parallel processing units. Software algorithms usually distribute multiple tasks into different cores to fully utilize its processing abilities. Multi-core solution can be regarded as a software level approach as the parallelism is mainly achieved by scheduling algorithm, which distributes workload to separate threads, pipelines and cores to fully utilize CPU and cache resources. At current stage, the throughput of this approach is still not fast enough for wire-speed DPI applications on core routers.

Guo et al. in [64] propose a parallel L7-filter architecture on a multi-core server. Their system uses a scheduling method to make CPU time more effectively distributed to the pattern matching threads, so as to eliminate the latency of scheduling stalls. However, the scalability of parallelism is restricted to the number of cores. Furthermore, their subsequent work in [18] uses an adaptive hash-based multi-layer scheduler to select a module from three levels, including core, pipeline and thread levels. Compared to [64], results in [18] show that the adaptive hash based multi-layer scheduler further improves L7-filter throughput by 59%. Besides, CPU utilization grows as the throughput of L7-filter increases. The grouping algorithm in [17] can also be deployed in multi-core architecture with a limited memory size. The algorithm keeps on adding new patterns with the least interactions with each other until the composite DFA exceeds the limit of its allocated memory. Then it proceeds to create new groups from remaining patterns. Each group forms a composite DFA, and each core can run one or more composite DFAs.

2.6 Low Power Design

Traditional network system does not pay enough attention to power consumption and network devices always run at full capacity all the time. Currently, most of energy efficiency research contributes for wireless communications due to their limited power supply. However, as Internet expands fast, the power issue cannot be ignored any more.

Preliminarily, energy efficient network schemes borrow the ideas of traditional low power techniques, which are mostly designed for battery-operated devices or components in wireless networks. Power consumption is determined by capacitance, voltage supply and running frequency etc, in view of this, voltage and frequency can be adjusted based on current workload to reduce the energy consumption without affecting system performance.

Network power consumption can be reduced at three levels: 1) component level; 2) system level and 3) network level. Firstly, component level power reduction resorts

to specific voltage and frequency scaling techniques, which have been proposed to reduce the energy consumption of particular components. Secondly, system level power awareness is derived from adaption to the fluctuation of traffic amount. For example, in a multi-engine system, real-time adaption schemes can be incorporated in the traffic scheduling engine by adjusting the number of active engines. On the premise of performance guarantee, some engines can be turned off for power savings and turned on for larger processing capability. Thirdly, network level power optimization analyzes the power consumption of a large number of linked network devices sharing network resources, the use of which depends on the cooperation of network protocols.

2.6.1 Power Measurement

System power consumption on FPGA etc. platforms is composed of static power consumption and dynamic power consumption, which consists of interface switching power and internal dynamic power. Static power is a static value and is the total leakage power dissipation of each cell in the component. As technology advances below 65nm, static power grows to be larger than the dynamic power, as observed in Figure 2-15.

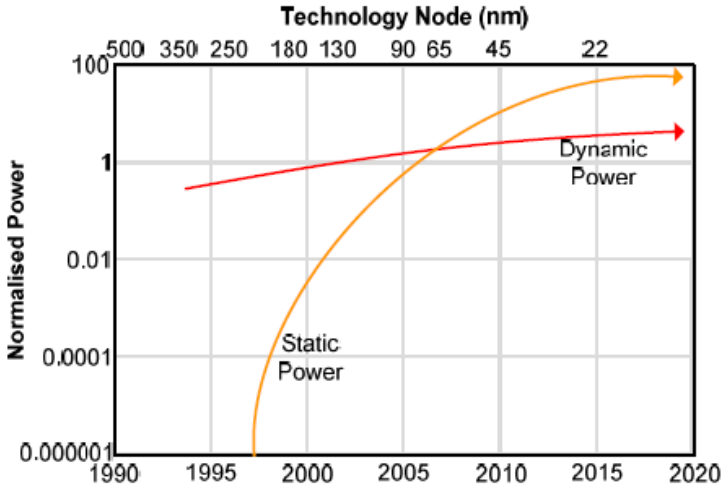


Figure 2-15 Trend comparison of static and dynamic power [65]

Dynamic energy dissipation per operation in a device is $E = C_{eff} V_{dd}^2$, where C_{eff} is

the effective switched capacitance and V_{dd} is supply voltage. For frequency f, the power dissipation for the operation is $P = C_{eff} V_{dd}^2 f$. The delay (t_d) on the critical path in a device determines the maximum frequency (f_{max}) and is defined in Equation 2-6 [66], where V_{th} is the threshold voltage, α is a technology dependent factor and k is a constant.

$$t_d = \frac{1}{f} = \frac{C_s V_{dd}}{I_{dsat}} \approx k \frac{V_{dd}}{(V_{dd} - V_{th})^{\alpha}} \tag{2-6}$$

Based on above equations, three conclusions can be deduced in theory: 1) if only V_{dd} is reduced, both energy and power can be saved at the cost of a longer delay; 2) if only f is reduced, power is lowered also at the cost of slowing down the circuit, but the energy per operation stays the same; 3) if V_{dd} and f are scaled in a coordinated manner (e.g. V_{dd} is reduced and f is increased), it is sensible that both energy and power can be decreased while maintaining the delay value. Practically, they are adjusted as a tradeoff.

The power consumption of a design based on certain platform (e.g. FPGA) can be estimated using software or hardware tools.

Software power calculation uses the activity and signal statistics analyzed during the compilation and synthesis step of the design. Before the power estimation, running frequency is an important parameter to be determined during the synthesis step, which should meet timing requirements of both adaptive clocking logic and a certain throughput of packet inspection engine. Most researches on power modeling use software power optimization techniques, instead of circuit-level low-power techniques related with electronic manufacturing. For example in [67], Franklin and Wolf developed an analytic performance model that captures a generic network processor's (NP's) processing performance and power consumption.

Hardware power estimation can be obtained as follows. Firstly, HDL program is simulated in Modelsim to verify and debug the code. Second, HDL level program is compiled and synthesized using FPGA tools such as Xilinx ISE or Altera Quartus to

generate the VCD (Value Change Dump) file. Third, based on VCD files, the power consumption is estimated by a power calculator functional block in ISE or specialized power estimator. In simulation, the total power is the sum of power consumptions for every major component under particular configurations, such as ALU and shifter, registers, queues etc. Power consumption can also be roughly estimated using Cacti [68], Wattch [69] etc. simulation tools.

Generally, the real power measurement is performed as follows. In order to obtain real power consumption data of a FPGA board, a PCI or PCIe bus extender is plugged into PC motherboard through PCI or PCIe interface, which is also used to connect to the FPGA board from slots with different voltage supplies, such as 5V, 3.3V, 1.2V etc. A voltmeter measures the accumulative power data; besides, due to the inaccuracy of manual readings, a USB or Ethernet based automation measurement device (e.g. Labjack [70]) is employed to join the bus extender to another PC for automatic recording of power consumption measurements. The final statistic power consumption is calculated as an average data of a group of sample values during a time period. A drawback of this approach is that it shows power consumption based on voltage categories; if there are several components on FPGA that are supplied with the same voltage level such as 3.3V, the display data is the sum power of all such components and cannot be further attributed to particular components.

For FPGA based design, specialized hardware tools integrate modules for power analysis and power optimization at different stages. For example, Synopsys Prime Power and Power Compiler [65] are used for power management. The Eclypse platform can give comprehensive support for advanced low power design. The power compiler can automatically make RTL and gate-level power optimization. Besides, there are other kinds of Power Estimation tools for general or specialized hardware platforms, including Altera PowerPlay Early Power Estimators (EPE) [71] and Quartus II PowerPlay [72], Xilinx XPower Estimator and XPower Analyzer [73] etc. power analysis tools.

Figure 2-16 shows an example of a popularly used power analysis tool, Altera

Quartus Powerplay. EPE provides approximate power information which is always used as pre-project estimation. When designing a Printed Circuit Board (PCB), the power supply should be able to satisfy the maximum sum power consumption of all the components.

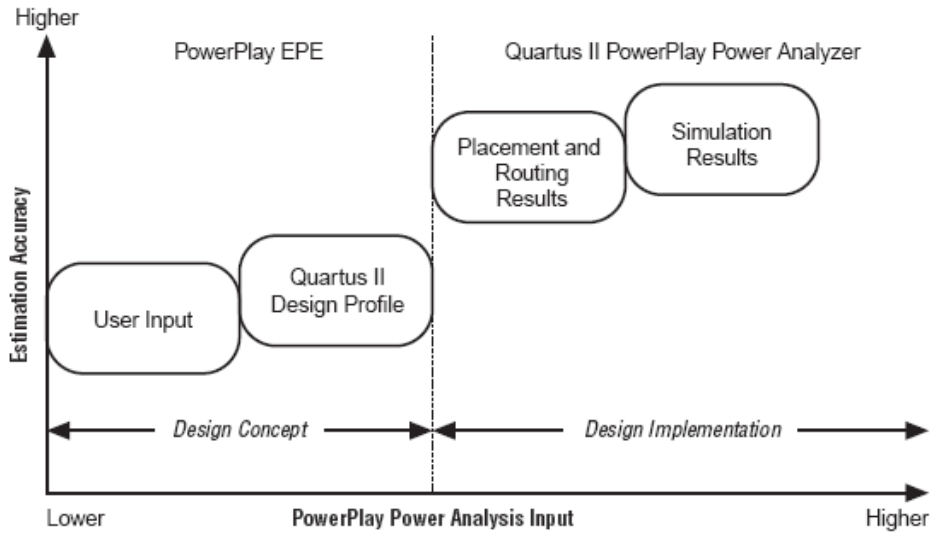


Figure 2-16 Altera PowerPlay power analysis [72]

2.6.2 Power Reduction at Three Levels

In recent years, along with a continuous growth of both energy cost and network energy requirement, academia and industry show big interests in energy efficient network technologies for ecological and economical reasons. Although the traffic load of networking system fluctuates from time to time, to be competitive, network design must be able to deliver peak performance while saving power. Power analysis and optimization techniques work at component level, system level or network level.

1) Component level power analysis and reduction techniques

At component level, since system components are not required to work at its peak capacity all the time, in order to be energy efficient, the system should have the ability to tune or switch its components to adjust its processing capability with current workload. Though the advance of Very Large Scale Integration (VLSI) or Ultra Large Scale Integration (ULSI) provides the ability of implementing rich

functionality on small die nowadays, leakage and dynamic power dissipations, arising from high frequency switching of the millions of transistors, directly bring larger challenges like battery, cooling and reliability.

The leakage current in nanometer devices has increased drastically due to reduction in threshold voltage, channel length, and gate oxide thickness. A large number of components in a highly integrated system are idle most of the time. The high-leakage devices and low activity rates lead to the growing significance of leakage power. Leakage power reduction depends on circuit level operations. Liao et al. in [66] use power gating to reduce micro-architecture level leakage power, which inserts sleep transistors between power supply and logic or memory circuits, and these sleep transistors are turned off to cut off from power supply when the circuits are idle. Compared to using sleep transistors with normal threshold voltage, Brooks et al. in [69] use Multi-Threshold CMOS (MTCMOS) sleep transistors with high threshold voltage to reduce leakage power. Virtual power Rails Clamp (VRC) in [68] further improves MTCMOS with data retention by inserting parallel diodes to sleep transistors.

2) System level power analysis and reduction techniques

At system level, power controller generally works by observing the network workload condition and controlling the running conditions. Since network is designed to guarantee the worst case performance, at system level under normal circumstances, it can be dynamically reconfigured to provide necessary capabilities with a minimum number of active components.

The reduction of energy consumption in hardware devices can be accomplished by adjusting its dynamic power without impacting peak performance of busy period. Dynamic power reduction techniques adapt its clock and voltage to make just in time completion with different workload for power efficiency. Common dynamic power reduction techniques include reducing switching power using intelligent clock gating, dynamic frequency scaling (DFS) and dynamic voltage scaling (DVS).

- **Clock gating**

Clock gating is used to reduce dynamic power consumption in synchronous circuits with a number of processing components. It works by switching off the clock to unnecessary components for low workload. When the system is not saturated, suspending part of modules will gain remarkable power savings. Practically, clock gating adds additional logic to a circuit so as to prune the clock tree. By disabling parts of the circuit so that their flip-flops do not change state, the switching power consumption goes to zero, and it only retains the leakage currents. The implementation of clock gating faces two challenges: 1) a small and accurate idleness-detecting logic, and 2) a gated-clock distribution circuit. The first should be able to stop certain clocks with small power and time overhead. The second requires small routing overhead and keeps clock skew under tight control [74].

For example, algorithm in [75] uses low power technique to turn off some Processing Elements (PEs) under low incoming traffic. It is based on the observation that for low traffic rate, less PEs are needed to process the traffic with low power consumption; and for high arrival rate, all the PEs are necessary. In order to turn off some processing engines, they use the clock-gating technique on PEs when the packet-processing requirement is low and re-enable the clocks when the processing demands grow higher. Considering the high cost of powering up the PEs, the motivation of using clock gating is to effectively turn the PEs to standby mode, rather than actually power them down completely. Furthermore, algorithm in [76] uses Deterministic Clock Gating (DCG) based on the fact that for many of the stages in a modern pipeline, a circuit block's usage in a specific cycle in the near future is deterministically known a few cycles ahead of time. [76] is also the first paper to show that a deterministic clock-gating methodology is better than a predictive methodology like pipeline balancing, which is essentially a methodology to gate the clock of unused components whenever a program's instruction-level parallelism is predicted to be low.

- **DVS**

DVS is a popular approach for energy reduction of integrated circuits. DVS has been used in general processors assisted by the scheduling algorithm of system tasks. The use of DVS on general purpose processors supports a range of operating voltages from full voltage to half of maximum voltage. A question for DVS is the determination of the lowest voltage bound for optimal energy efficiency. The voltage limit depends on two features, the power and delay tradeoff at low operating voltages, and the workload characteristics of the processor.

- **DFS**

DFS is another power conservation technique that works similarly as DVS. Compared with DVS, DFS is more stable because of its simpler implementation, and it needs only a few clock cycles to switch on and off. DFS reduces the number of instructions a processor can issue in a given amount of time, and it lowers the performance. Therefore, DFS is used when the processor is not saturated.

- **Comparison of above three power reduction techniques**

DFS allows devices to dynamically adapting their speed so as to increase power efficiency of their operation. DVS varies processor voltage under algorithm control to dynamically meet performance requirements. Clock-gating technique saves dynamic power consumption, but the clock-gated components still consume certain leakage power. Besides, clock gating clearly affects throughput due to the wake up time before the system works normally. Comparatively, DVS and DFS do not turn off components but supply them with scaled clocks and voltages. Both DVS and DFS are used in computer processors to prevent processor overheating; the overheating requires effective cooling measures or it results in system crash. On the other hand, reducing voltage or frequency supply also causes system instability.

3) Network level power analysis

At network level, all devices connected together through network belong to an integrated system. The cumulative energy for Internet infrastructure increases fast for two reasons: 1) the expansion of the Internet and 2) the increase in the diversity and

complexity of applications as well as their level of performance supported by packet processing systems. Energy-aware platforms and green network protocols develop accordingly. Besides power savings on a single router, multiple routers along the forwarding link can cooperate together with certain power plan and notify each other with incoming traffic conditions. At network protocol level, power aware routing algorithms are applied to determine the end-to-end routing paths and the packets are sent through a link with minimal energy consumption. It switches among different routes during busy or idle period within the guarantee of expected performance.

Energy demand of the Internet will keep growing fast if power is not given enough consideration in network processing. Chapter 5 focuses on the research of system level power efficiency. Network level power-awareness with modified protocols will be studied in the future work.

2.7 Summary

DPI is key component in NIDS for network security. DPI performs pattern matching on packet payload, through fixed string matching like Bloom filter etc. algorithms and regular expression matching using DFA or NFA.

Basic DFA and NFA algorithms have some drawbacks. For DFA, certain features in substrings lead to potential state explosion, such as wild cards and counting constraints. In order to be memory and bandwidth efficient, some improvements are proposed in research including pattern rewriting, pattern groupings, pattern splits and multiple stride DFAs. Since regular expression matching requires a large amount of memory, compression is needed to optimize memory utilization. Default transition compression is based on the property of locality in DFA transitions, and it eliminates similar transitions at different nodes with the same input. The alphabet compression and the use of separate fast and slow path all contribute to memory optimization. Besides, based on DFA and NFA, there are novel kinds of state machine structures by adding labels, counters, or using a combination of DFA and NFA advantages.

Bloom filter is an efficient data representation method and has been widely used in database and network applications. Computer and network applications largely involve information representation and lookup. Therefore, compressed storage structures and fast query methods can significantly improve processing capability. A Bloom filter uses a number of hash functions to map each input string to a hash array, the input string is considered as a pattern if all the mapped entries are equal to one. It can be safely filtered if at least one of the entries is zero. But the matched results include false positives. There are many variations based on regular Bloom filter, such as Space-code Bloom filter, Spectral Bloom filter etc. They improve regular expression matching in reducing memory consumption or reducing false positive rate.

To keep up with network speed, hardware accelerators using TCAM, FPGA or multi-core network processors are needed. The major challenge is the tradeoff between performance, flexibility and device cost. Internet power consumption grows fast in recent years. Low power techniques target at three levels: component, system and network levels. In general, low power network design saves the power wasted on idle or low link rate period while maintaining required network performance.

Chapter 3:

Memory Reduction of DFA Algorithms

This chapter analyzes the redundancies in DFA transitions for memory compression. In order to reduce the memory consumption of the DFA, two improved DFA methods, Extend-D^{2}FA and Tag-DFA, are proposed to exploit more than one kind of redundancy in DFA transitions, and are compared with a well-known algorithm D^{2}FA.

3.1 Introduction

The rapid growth of the Internet has made it increasingly vulnerable to various threats and attacks, such as intrusions, worms, viruses and spam. DPI has been adopted by enterprises to defend against all kinds of attacks. The rich expressiveness of regular expressions makes it dominant for DPI implementation in NIDS nowadays. Among popular open source NIDS softwares, Snort [9] has a large number of regular expression based signatures and all of the rules in Bro [10] are written in regular expressions, L7-filter [11] also exploits on regular expressions for protocol identification.

Regular expression based approaches have significant memory complexity. NFA has multiple concurrent active states and it brings unpredictable bandwidth costs while DFA needs only one state traversal to process one character at the cost of larger memory consumption. Comparatively, since it is time consuming to track multiple states simultaneously, so far there are not many fast NFA based implementations for NIDS. On the other hand, DFA based approaches are suitable for both software and hardware implementations. Moreover, DFA can be easily updated by integrating new regular expressions except the challenge of state explosion.

There are some situations like counting constraints with the wild cards that might

cause state blow-up of DFA. Although in most cases, the number of states in DFA is comparable to that of its corresponding NFA, given an *n*-state NFA, the upper bound of the corresponding DFA is 2^n states. For instance, Snort [9] has more than 7000 signatures and its memory size of naive DFA-based approach is exponentially scaled in the worst case, for example, the Snort HTTP rule set will need more than 15GB memory using simple DFA [22].

DFA algorithm is fast but needs large memory size. In the storage of DFA structure, states are connected through transitions, and one state is associated with a number of inward and outward transitions. Transitions have redundant information and can be further compressed. D^2FA [16] is one of the most famous approaches that have been proposed for transition compression. By the use of default transitions, D^2FA can significantly reduce memory consumption, but it is at the cost of multiple accessing times of DFA per input character. CD^2FA [19] is proposed using recursive content labels to reduce the diameter bounds of D^2FA to 2 with one 64-bit wide memory access. M. Becchi et al. in [46] propose to limit the bound of the number of default paths in D^2FA, which improves the worst case performance of D^2FA.

In this chapter, my work is based on a classical transition compression algorithm D^2FA for DPI. Another transition compression algorithm, CD^2FA, is not used in this thesis since it introduces labels with matching indices, which complicates its creation and the analysis of transition redundancies. Through our analysis, we find that D^2FA has some blind spots on certain rules and do not comprehensively exploit all the possible redundancies in DFA. In order to be time and memory efficient, this chapter focuses on DFA memory compression and its fast inspection. There are three contributions in this chapter.

Firstly, the redundancies in DFA transitions are analyzed for memory compression.

Secondly, the classical D^2FA is extended to Extend-D^2FA by introducing a base table, which can eliminate one more kind of redundancy than D^2FA. Extend-D^2FA solves the blind-spots of certain conditions in D^2FA compression. Experiments show

that Extend-D^2FA can achieve a larger compression ratio of transitions in DFAs with the same number of default transitions as D^2FA. Extend-D^2FA has small improvement for the rule sets that D^2FA have compressed pretty well.

Thirdly, an improved DFA structure called Tag-DFA is proposed by introducing some new states with tags. The new states represent compressible information among multiple states. Meanwhile, tags distinguish different kinds of redundancies in the new states. Tag-DFA can obtain the next state with only one hop. Overall, Tag-DFA avoids the problem of multiple default paths in D^2FA and it deals with all the four kinds of redundancies for better memory efficiency. The evaluation shows that Tag-DFA can save more than 90% of memory consumption compared with the original DFA implementation.

This chapter does not give theoretic analysis since the compression methods are based on exploring compressible redundancies of DFA transitions observed from examples. Snort, Bro and L7-filter are three popular NIDS tools, and their rules include different kinds of patterns for the well-known protocols and web attacks that are summarized by network vendors. The DFAs and NFAs in this chapter are generated by part of Snort and Bro rules, which are included in DFA generation tool, Regex [43]. The matching data is randomly generated with certain matching probabilities.

3.2 Memory Analysis of DFA and NFA

Both the DFA and the NFA algorithms are widely used in DPI solutions. To show the difference of the two schemes, their memory size and processing speed are compared in this section before my research on DPI. For my implementation, the fast and deterministic speed is more important, thus the DFA algorithm is chosen. Considering its large memory size, I focus on DFA transition compression as my method of DFA memory reduction, which will be presented in later sections of Chapter 3. For the comparison of DFAs and NFAs, since other related publications on state automata optimizations mainly compare the size of their own algorithms with the

size of traditional or table compressed DFA or NFA, they are not referenced in this section.

In high-speed network applications, DFA is widely used to represent regular expressions due to its low and deterministic bandwidth requirement. With one state traversal per character, DFA has predictable running time, and it maintains only one active state to reduce the complexity in "per-flow" network links. However, comparing the DFA and the NFA, the expensive memory requirement restricts DFA's usage for complicated rules and the combination of regular expressions.

This section analyzes the characteristics of DFA and NFA, especially the memory consumption of DFA, and also explores the following opportunities in DFA organization for memory savings including: 1) a pre-filter for a smaller size of active memory; 2) grouping of characters; 3) the use of cache; and 4) the use of different memory encoding methods.

3.2.1 Characteristics of DFA and NFA

The memory and bandwidth requirements of DFA and NFA are compared using RegEx [43]. The experiment uses datasets of *Snort24.re* and *Bro217.re* in RegEx [43], which are selected from Snort [9] and Bro [10] patterns. For simplicity, the patterns used in the experiment are only a very small part of Snort and Bro rule set. *Snort24.re* is a representative set of complex patterns and *Bro217.re* belong to normal patterns.

Table 3-1 compares the memory size of DFA and NFA. In comparison, the intricate 24 rules from Snort generate a small NFA, which is converted to a huge DFA; and the simpler 217 rules from Bro generate NFA and DFA of similar sizes. The states and transitions of *Snort24.re* DFA are more than 13 and 5 times as those of their corresponding NFA.

Table 3-1 Memory size comparison of DFA and NFA

Rule set	Traditional DFA		NFA		Comparison (DFA/NFA)	
	States	Transitions	States	Transitions	States	Transitions
Snort24.re	8335	18805	603	3610	13.82	5.20
Bro217.re	6533	11247	2131	5423	3.06	2.07

While NFA uses a much smaller number of states and transitions, it requires larger bandwidth, as shown in Table 3-2 for the average number of active states during the parsing of four trace files. The trace files are randomly generated by RegEx [43] based on rule set *Bro217.re* with four degrees of the matching probability (P_m), 35.0%, 55.0%, 75.0%, and 95.0%. The number of active states increases with the matching probability. DFA requires only one active state while NFA requires about 7 active states on average and this slows down speed, which is a serious defect of NFA.

Table 3-2 Number of active states in NFA

Active states	*Matching probability with Bro217.re*			
	35.0%	55.0%	75.0%	95.0%
Average	6.9	7.3	7.5	10.0
Max	11.0	13.0	14.0	16.0

Next the average number of state traversals per input is compared. The trace files are randomly generated with four degrees of matching probabilities (P_m), based on the same two rule sets as above. As shown in Table 3-3, the DFA of dataset *Snort24.re* shows a smaller number of state traversals than those of NFA, which is about 81% and 33% compared to NFA with input traffic of matching probabilities 35% and 95%. Under the matching probability of 35%, the average number of state traversal for the NFA of *Snort24.re* is 2.38 and the average number of state traversal for the NFA of *Bro217.re* is 12.87. Comparing the two groups of datasets, *Snort24.re* has less rules (24 compared to 217), and according to Table 3-1, the NFA of *Snort24.re* has more transitions and states per rule than that of *Bro217.re*. With an input character, the larger uncertainty of *Bro217.re* NFA leads to more steps of state traversals.

As there is more than one active state in NFA during the matching process, the average number of state traversal per input increases as P_m increases. DFA uses extra traversal back to the initial state for a non-match in implementation, thus the

difference of the number of state traversals is getting smaller as P_m increases. The larger memory size of DFA also guarantees deterministic transitions that contribute to a much faster and more predictable matching speed.

Table 3-3 Comparison of average number of state traversals per input

Automata	*P_m with Snort24.re*				*P_m with Bro217.re*			
	35.0%	55.0%	75.0%	95.0%	35.0%	55.0%	75.0%	95.0%
DFA	1.95	1.82	1.57	1.20	1.41	1.47	1.36	1.25
NFA	2.38	2.63	2.82	3.62	12.87	13.33	13.50	16.96
Comparison	81.9%	69.2%	55.7%	33.1%	11.0%	11.0%	10.1%	7.4%

3.2.2 Pre-filter for Smaller Active Memory

The use of pre-filter can significantly reduce the demanded sizes of DFA and NFA, since network traffic is mostly benign with only a small percentage malicious. A majority part of memory space in DFA or NFA, therefore, is seldom visited. In fact, matching probability of normal network trace is smaller than 1%. This experiment simulates traces under extreme conditions to show the states access rate. With the same trace files as above, Table 3-4 gives the percentage of state traversal in DFA and NFA.

Table 3-4 Percentage of traversed state

Automata	*P_m with Snort24.re*				*P_m with Bro217.re*			
	35.0%	55.0%	75.0%	95.0%	35.0%	55.0%	75.0%	95.0%
DFA	1.4%	2.4%	3.9%	4.3%	15.0%	24.1%	40.5%	84.0%
NFA	20.2%	34.7%	56.4%	61.4%	34.3%	55.7%	93.3%	100%
Comparison	6.9%	6.9%	6.9%	7.0%	43.7%	43.3%	43.6%	84.0%

Similar to Table 3-3, the different representative characteristics of two rule sets generate different results. In comparison, more states in the DFA and NFA of *Bro217.re* have been accessed during the matching process. For instance, under the matching probability of 35%, the traversal percentage of *Snort24.re* DFA states is 1.4% and that of *Bro217.re* DFA states is 15%; and under the matching probability of 95%, only 4.3% of *Snort24.re* DFA states have been visited; on the other hand, 84% of *Bro217.re* DFA states have been visited. The difference for the two rule sets is due to the reason that a large part of *Snort24.re* DFA is built for special circumstances and is

seldom used for normal traffic. The redundancy of *Snort24.re* DFA can be seen from Table 3-1, the *Snort24.re* DFA states is 13 times as its corresponding NFA while the *Bro217.re* DFA states is 3 times as its corresponding NFA. This also explains the difference that the percentage of *Snort24.re* DFA traversed states is about 7% of its corresponding NFA, but that of *Bro217.re* is 43.7%.

The average traversal percentage of DFA states is small. Therefore, the historically popular accessed states can be extracted as a pre-filter in DPI, where confident non-malicious traffic can be let through and suspicious traffic is sent for further matching. Next the pre-filter structure is presented with a smaller rule-set for prefix matching.

Figure 3-1 shows the working procedure of a DPI system with pre-filter for the pattern training and the pattern matching. Pattern compiler module generates NFA from pattern rule-sets and converts NFA to DFA for pattern matching engine, which uses a small pre-filter to scan all the packets and a verifier to check the initially matched ones. Normal packets pass while malicious packets are removed. The detected patterns, protocols or virus etc. are kept is a log file. The log file is also analyzed for feedback and refinement to a DFA optimizer in the compiler module.

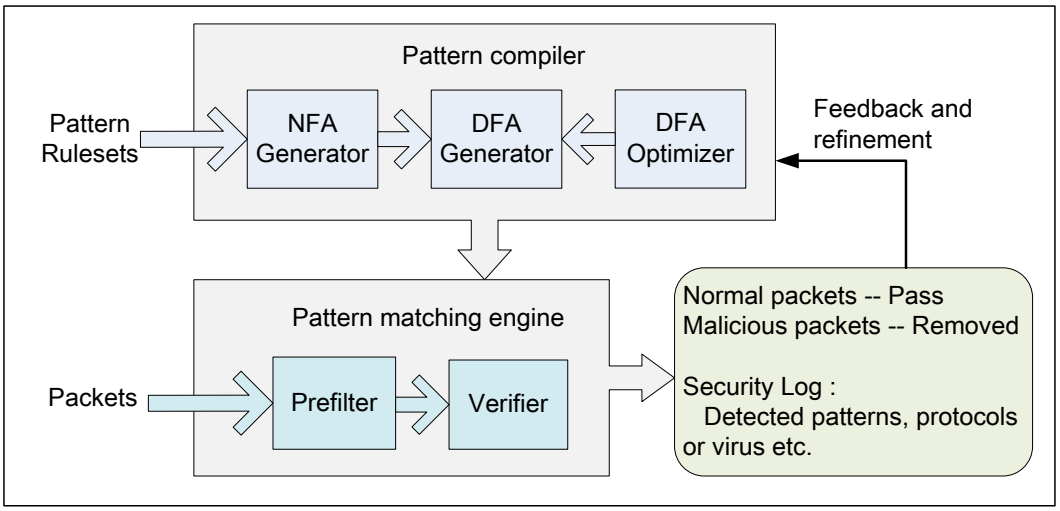


Figure 3-1 DPI working procedure with pre-filter

In particular, Figure 3-2 shows the structure of a pre-filter, where the packet

header and payload are processed by a header checker and a payload filter, respectively. Experts in network security extract header rules and pattern rules from Internet attacks that have been detected. The payload is filtered with a small part of pattern prefixes for efficient initial inspection. The preliminary filtered results are classified into three categories: 1) malicious packets to be discarded by a discarder; 2) benign packets to be forwarded by a forwarder; and 3) suspected packets to be further verified using full inspection. Generally, a large percentage of network traffic will be identified as benign packets to bypass the subsequent verification. Thus the significance of two-stage pre-filter scheme is the faster overall speed with smaller active memory. The chances of state explosion can be largely reduced since the slow path rules can be put in the second stage.

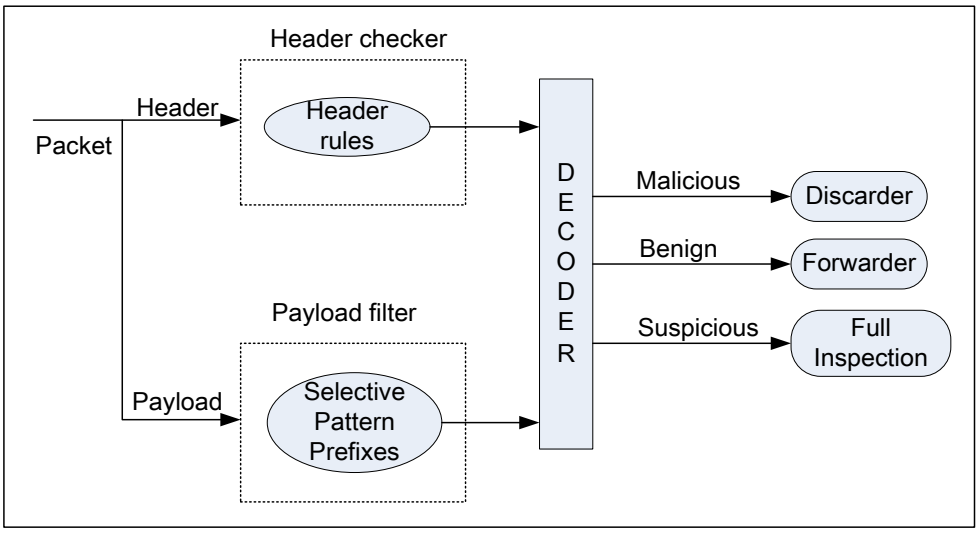


Figure 3-2 A particular structure of a pre-filter

Concerning the pattern matching engine, Figure 3-3 shows a two-stage pattern matching system with a pre-filter and a number of verifiers in parallel. The pre-filter is constructed from fast path prefix in DFA and the multiple verifiers are constructed from the slow path suffixes. The matching separation can be packet-based or byte-based, the latter one responds faster to potential attacks but with more triggers to slow path. Principally, the slow path, constructed with suffixes, only inspects the flows that have been matched by the fast path, constructed with prefixes. Thus the

bigger slow path automata can be stored on a secondary memory and sleep most of the time.

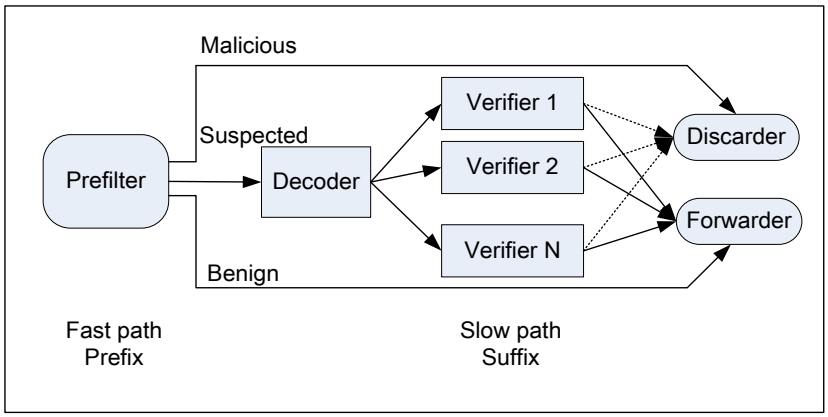


Figure 3-3 a two-stage pattern matching system

3.2.3 Grouping of Characters

Grouping of characters can effectively reduce the memory consumption of DFA. In DFA generation stage, characters are grouped by analyzing the initial DFA state machine. Two symbols will fall into the same group if they are treated the same way in all DFA states, that is, they transfer from the same source states to the same target states. The grouping of characters firstly chooses the patterns with the least interaction with each other, and adds other patterns in this group. In this way, the DFAs for separate groups have no states explosions, but their combination might cause exponential scale of states. Besides, some ASCII characters rarely appear in DPI patterns and they can group as one class in the state machine.

In experiment, the transitions in the above DFAs are analyzed as inward or outward transitions for each state. After character classification, statistics show that over 80% of states have only one inward transition. As shown in Table 3-5 for the distribution of outward transitions that is the number of output character groups. *Snort24.re* has 58 character classes, and more than 96% of its DFA states have only one outward transition, and 95.5% of states has only 1 outward transitions. Regarding *Bro217.re* patterns, it has 111 character classes after character classification. Over 95%

states have only one outward transition; over 76% states have one or two inward states. This feature of DFA is utilized for compact memory representation.

Table 3-5 Proportions of transition distribution after grouping

Rule set	Classes	Inward transitions				Outward transitions		
		1	2	3	Others	1	2	Others
Snort24.re	58	80.2%	9.2%	4.5%	6.1%	96.1%	0.8%	3.1%
Bro217.re	111	43.2%	33.7%	8.4%	13.7%	95.5%	4.4%	0.1%

3.2.4 Cache Consideration

Similar to the traditional cache in computer architecture, when the huge structure of DFAs cannot be entirely stored in fast memory, a small on-chip memory can be used as cache in complement to the storage of the whole DFA.

The idea of the cache design is to obtain smaller processing time through a higher cache hit rate. The cache hit rate also depends on cache organization and cache replacement policy, such as cache line size and cache associativity. The simulation of cache effectiveness works by comparing the issued address with the cached addresses. A state in an automaton is converted in a particular way to a specified address in cache. A cache position is initially invalid and set to be valid if a new state is brought in. Suppose the cache associativity is *n*, the traversal of a state machine first checks if it exists in any of *n* mapped positions. If the state exists in the cache, it produces a cache hit; or else, it replaces an invalid one among *n* mapped positions. If all are valid, the new state is cached with eviction of one of them.

A cache hit saves a large percentage of time compared to a cache miss. Consider two memory devices, memory D_1 with memory size M_1 and access frequency f_1, and memory D_2 with memory size M_2 and access frequency f_2. Content of D_2 is updated by removing the original content and storing a part of D_1 of size M_2 to D_2, and the time cost by one update is T_u. If w cache misses occur during the processing of an address text $A_1 \ldots A_n$, the total time cost is shown in Equation 3-1.

$$T = (n-w)\frac{1}{f_2} + w\left(\frac{1}{f_1} + T_u\right) = w\left(\frac{1}{f_1} - \frac{1}{f_2} + T_u\right) + \frac{n}{f_2} \quad (3\text{-}1)$$

In order to minimize T for an input of given length, we need to minimize the number of cache misses w. The key point is that if one cache miss occurs, it can be inferred that the proper content of D_1, once migrated to D_2, will reduce the cache misses in the near future. That is, every time we update D_2, it is desired that the new content should contain the addresses which would be accessed in the next a few addresses with the largest possibility.

The motivation of caching in DFA implementation is to make use of the physical localities of logical neighbors. States are stored in memory according to their sequence numbers. When one state is matched, the system brings in a block of states around this state. If the latter matched state is close to this state, it can be directly found from cache, which is a cache hit. Original DFA does not consider cache in its organization. Larger cache hit rate can be achieved by reordering state sequences, so that adjacent states are more likely to accessed one after another.

The use of cache accelerates DFA processing. Moreover, cache consideration is orthogonal to other DFA algorithms and can be used in combination. Taking into account of slow path and fast path in the pre-filter scheme, different cache sizes can be assigned to each of them. In addition, different engines in a parallel system can share a common cache, or use a separate cache table within the engine.

3.2.5 Memory Encoding Construction

In addition to the full representation with 256 ASCII characters specified, there are three other methods to encode states and transitions of state automata, including: 1) linear encoding; 2) bitmapped encoding and 3) address indirection also called content addressing. These three methods can be applied to either DFA or NFA. Compared to full representation, these encoding schemes save a large percent of memory size. In particular, they allocate memory differently and thus differ in memory size and the

number of memory accesses required for each state traversal.

- **Linear encoding**

Using linear encoding, a state with *l* labeled transitions is encoded through (*l+1*) 32-bit words, among which the first word represents the default transition, and the other words represent the remaining transitions, as shown in Figure 3-4. In particular, each word has the following fields: 1) one bit indicating whether this transition is the last one within the state; 2) 8-bits represents the input character upon which the labeled transition will be taken and 3) the remaining 23 bits devoted to the next state address. A state traversal starts from the first word, and involves going through the transitions in sequence until the one matching the input character is found or it is the last transition.

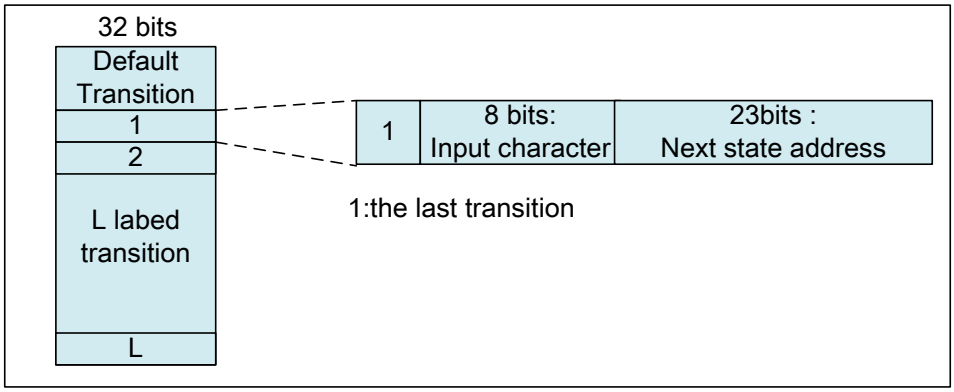


Figure 3-4 Linear coding for a state

- **Bitmap encoding**

Bitmap has a reasonable upper bound on the number of memory accesses needed to process a character. As shown in Figure 3-5, with the bitmapped encoding method, each state is encoded through a bitmap of 256 bits corresponding to ASCII code, and a sequence of (*l+1*) memory words, each word representing a next state pointer. During state traversing, it first analyzes the bitmap; if there is "0" in the position of input character, it directs to the default transition. Different from the table in Figure 3-4 for linear encoding, the entries in Figure 3-5 do not store tags for input characters, because the position in the bitmap determines the corresponding characters by counting the number of nonzero entries ahead of it. Bitmaps scheme provides

guaranteed deterministic and acceptable memory accesses.

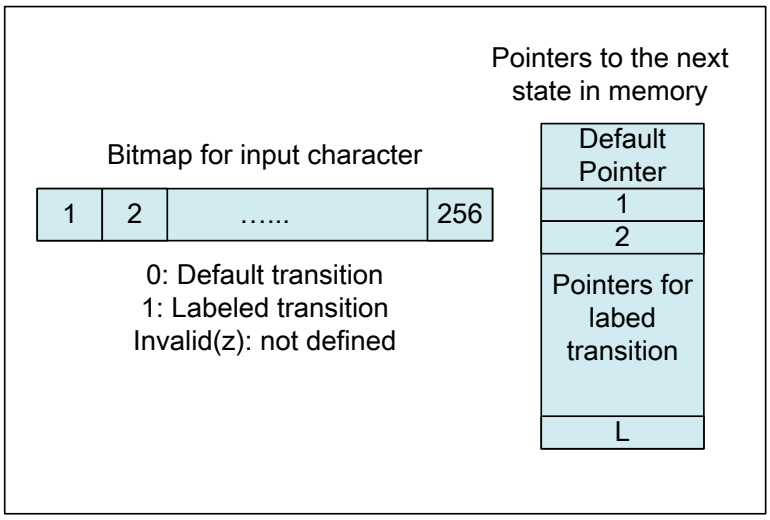


Figure 3-5 Bitmap memory structure

- **Indirect addressing**

Indirect addressing can be used to further reduce the memory bandwidth requirements. For each state traversal, indirect addressing allows a single memory access and a hash computation for the next address. Each state has an identifier consisting of the list of characters upon which there exists an outgoing transition, and a set of bits called state discriminator. State discriminators ensure that all state identifiers are different, even for states having labeled transitions on the same set of characters.

Take an example in Figure 3-6, if the state identifier is 32-bit, an address pointer is 8-bit wide and the state discriminator is 8-bit wide, then states with no more than three outgoing transitions can be represented by state identifiers, while other states should be fully represented with each input character specified. Similarly, a 64-bit identifier can represent a state with at most seven outgoing transitions. Through indirect addressing, if the current state address is stored at *state_address*[*state*] and a next state address is 4-byte wide, the address for *i*-th transition of the state can be found at *result*[*i*] =*state_address*[*state*]+*i**4.

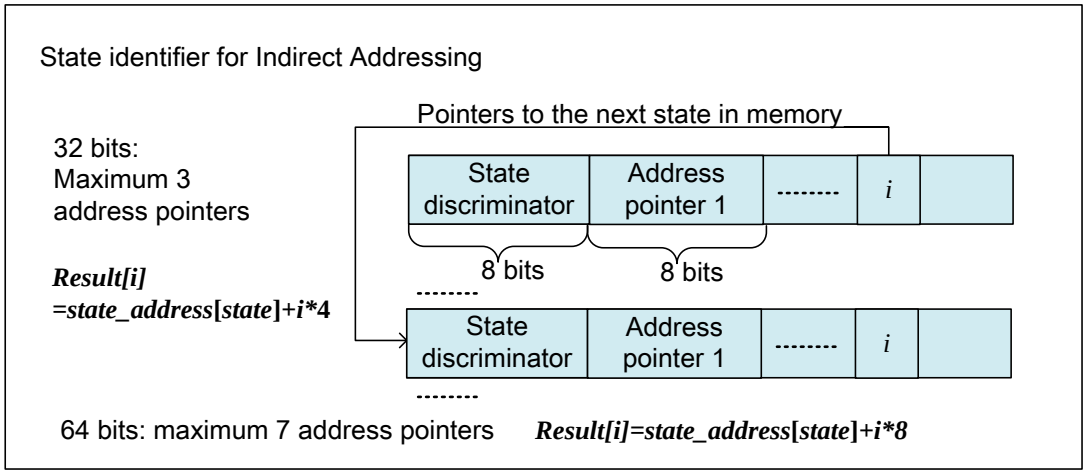


Figure 3-6 Indirect addressing encoding structure

- **Comparison of encoding methods**

The full representation of a state has 256 next state pointers for each possible input and it can find the next state directly with any inputs; comparatively, the three methods above tightly represent states in a state machine, while it takes longer time to find a transition. As a tradeoff, the implementation of memory layout always associates with a threshold, which restricts the number of transitions in a state. For instance, the threshold for 64-bit indirect addressing is 7. More than 99% of states fall within this threshold. If a state has more transitions than the threshold, this state should be fully represented with each input character specified.

- **Experiment results**

Using Regex [43], we compare memory size for three layouts of DFA and NFA of *Snort24.re* and *Bro217.re* with certain thresholds, as shown in Table 3-6 and Table 3-7. If the number of transitions of a state is above the threshold, this state will be fully represented and placed in base memory. If the number of transitions of a state is under thresholds, this state will be represented in certain methods, i.e. linear, bitmap and indirect addressing, starting from the offset for base states (e.g. 7KB or 9KB). It can be seen that although only a small number of states fall into base memory, full representation occupies much larger space. Takes linear NFA of *Snort24.re* as an example, 12 out of 603 states fall into base memory, and the 12 base states takes 9KB while the rest of 591 states take 6KB (that is 15KB-9KB = 6KB).

Table 3-6 Memory sizes of three encoding methods for *Snort24.re*

Memory Encoding	Threshold	DFA: states 8335		NFA: states 603		
		Base States	Memory	Base States	Offset	Memory
Linear	50	39	120KB	12	9KB	15KB
Bitmap	50	39	183KB	12	9KB	20KB
Indirect_add32	3	619	679KB	61	31KB	63KB
Indirect_add64	7	477	1081KB	40	42KB	85KB

*The offset size of DFAs is omitted in this table

Table 3-7 Memory sizes of three encoding methods for *Bro217.re*

Memory Encoding	Threshold	DFA: states 6533		NFA: states 2138		
		Base States	Memory	Base States	Offset	Memory
Linear	50	2	69KB	9	7KB	21KB
Bitmap	50	2	118KB	9	7KB	37KB
Indirect_add32	3	199	261KB	89	7KB	100KB
Indirect_add64	7	38	207KB	26	14KB	76KB

*The offset size of DFAs is omitted in this table

From Table 3-8, indirect content addressing takes larger memory than linear or bitmap representation methods. On the other hand, indirect addressing works faster, as reflected in the number of clock cycles per input. For *Snort24.re*, indirect addressing needs 23 while linear and bitmap need 8 to 9 memory accesses.

Table 3-8 The average number of memory accesses for each input character

P_m in trace	*Snort24.re*				*Bro217.re*			
	Linear	Bitmap	Indirect _add32	Indirect _add64	Linear	Bitmap	Indirect _add32	Indirect _add64
35.0%	8.63	7.28	2.38	2.38	27.90	31.07	15.05	18.30
55.0%	8.22	7.65	2.63	2.63	29.20	31.80	15.70	19.74
75.0%	8.17	8.05	2.82	2.82	28.78	29.34	15.35	18.47
95.0%	8.41	8.86	3.26	3.26	33.36	34.34	18.72	25.00
Average	8.36	7.96	2.77	2.77	29.81	31.64	16.21	20.38

Next the representation methods are associated with cache. From the cache hit rates in Table 3-9, linear encoding method has the largest cache hit rate, followed by bitmap encoding. The indirect addressing does not work well with the cache module in Regex [43].

Table 3-9 Cache hit rate of three encoding methods

P_m of trace	*Snort24.re*				*Bro217.re*			
	Linear	Bitmap	Indirect _add32	Indirect _add64	Linear	Bitmap	Indirect _add32	Indirect _add64
35.0%	97.61	90.28	71.3	96.32	94.58	94.59	49.73	79.14
55.0%	95.12	86.32	69.16	92.99	93.06	90.72	42.46	75.96
75.0%	92.2	88.38	69.8	91.29	86.03	78.72	38.67	68.08
95.0%	93.45	85.18	72.69	88.76	74.02	65.26	28.18	61.97
Average	94.59	87.54	70.74	92.34	86.92	82.32	39.76	71.29

In comparison, indirect addressing requires the largest memory but its processing speed per input is the fastest with a smaller number of memory accesses per input. On the other hand, linear and bitmap encoding are simpler and have larger cache hit rates.

3.3 Redundant Information in DFA

A key issue for DFA compression is to identify which parts in DFA storage are redundant and can be compressed. In particular, we focus on transition compression. D^2FA is the classical transition compression algorithm for DPI. Another algorithm, CD^2FA, is not used in this thesis since CD^2FA introduces labels with matching indices, which also complicates the analysis of transition redundancies. D^2FA and CD^2FA reduce one type of redundancy (i.e. TDS as introduced below).

We first point out the weakness of D^2FA [16] compression for certain kinds of regular expressions. After that, we summarize four groups of transition redundancies in the original DFA structure, and analyze how to compress some of the transitions by exploring inter-state or intra-state similarities. Then we propose two improved algorithms based on D^2FA: Extend-D^2FA and Tag-DFA, as explained in sections 3.4 and 3.5.

3.3.1 DFA Structure

Figure 3-7 shows a traditional DFA structure and an example of basic DFA expression. Each state stores next hop state pointer for alphabet characters and has 256 next states and transitions for ASCII alphabet, which leads to large redundancy.

Instead, most DFAs generators perform classifications for transitions with the same next hop. In Figure 3-7, based on the regular expression, 256 ASCII characters are divided into 4 classes, which efficiently reduces the size of DFA memory. In experiment on L7-filter [11], there are only 14.625 transitions on average after character classification.

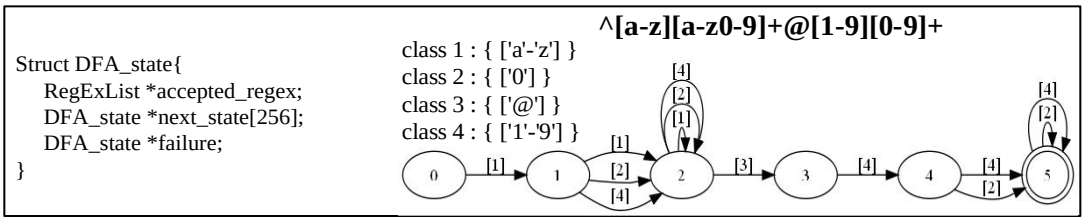


Figure 3-7 Traditional DFA state and an example of DFA classification

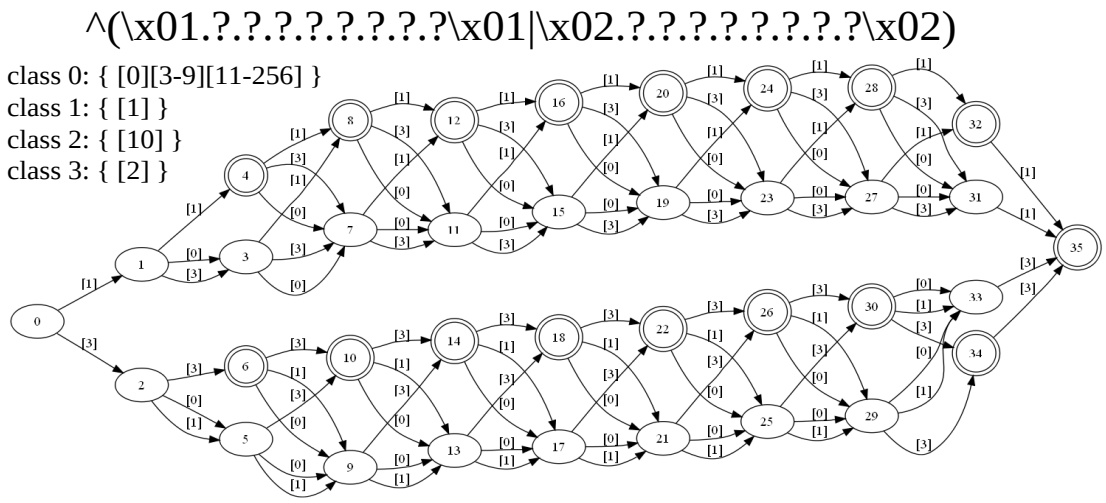
However, memory cost is still very large even after using this method. Take the 104 protocols in L7-filter [11] as an example, they have 11499 states, and 791153 transitions without any combinations. Even with a conservative estimation, it still requires 12.6Mbits (791153×16bits), and it does not consider the geometrical series increase of states as well as the transitions for combination of multiple rules. Therefore, DFA compression becomes very important, and we focus on transition compression for DFA based on D^2FA [16].

3.3.2 Weakness of D^2FA Compression

D^2FA [16] is one of the most famous approaches that have been proposed for transition compression. As briefly presented in Section 2.3.4, it develops an algorithm to compress DFA using default transitions, which can reduce memory consumption at the cost of potential multiple memory accesses for each input character.

However, D^2FA has some fatal blind spots for some DFAs constructed in our experiments. As an example, we choose part of a regular expression for "*skypeout*" protocol in L7-filter [11],"*^(\x01.?* $^{(8)}$ *\x01|\x02.?* $^{(8)}$ *\x02)*", the matching of which requires 8 arbitrary characters between two "1"s or "2"s, as shown in Figure 3-8, this regular expression is translated to a DFA with 36 states. But for each state, there are

only two different corresponding parent states, except the initial state and the final state. For instance, the parent states for State 12 are State 7 and State 8. In other words, for each state, there is at most one state which has transitions towards the same next state. The memory reduction of D^2FA relies on compact representation of states with the same next state, e.g. State 7 and State 8, or State 11 and State 12. In this case, D^2FA does not compress transitions of different states towards the same next hop states, and the upper bound of compression ratio of D^2FA is 50%.



*class 2 is generated for the specification character "10" in other parts of "*skypeout*"

Figure 3-8 An example of DFA for a regex in "*skypeout*"[11]

Take another example, the L7-filter rule for "*edonkey*" (i.e. a P2P file sharing network) [11], part of which follows the way as "*^[\xc5\xd4]… ([\x01\x02...$)*", will produce 4110 transitions; however, D^2FA can hardly reduce its memory usage. No transitions from different states point to the same next state in this DFA. Therefore, D^2FA has poor compression effects for this kind of explosively expanding DFA. In fact, memory waste is still very large in this kind of DFA. Thus we need to develop other approaches to compress them.

3.3.3 Classifications of Compressible Transitions

In this section, we analyze which parts of DFA can be further compressed. As shown in Figure 3-9, based on the inter-state and intra-state characteristics, the

compressible DFA transitions are categorized into four groups: TDS, TSS, TLCS and TLNS.

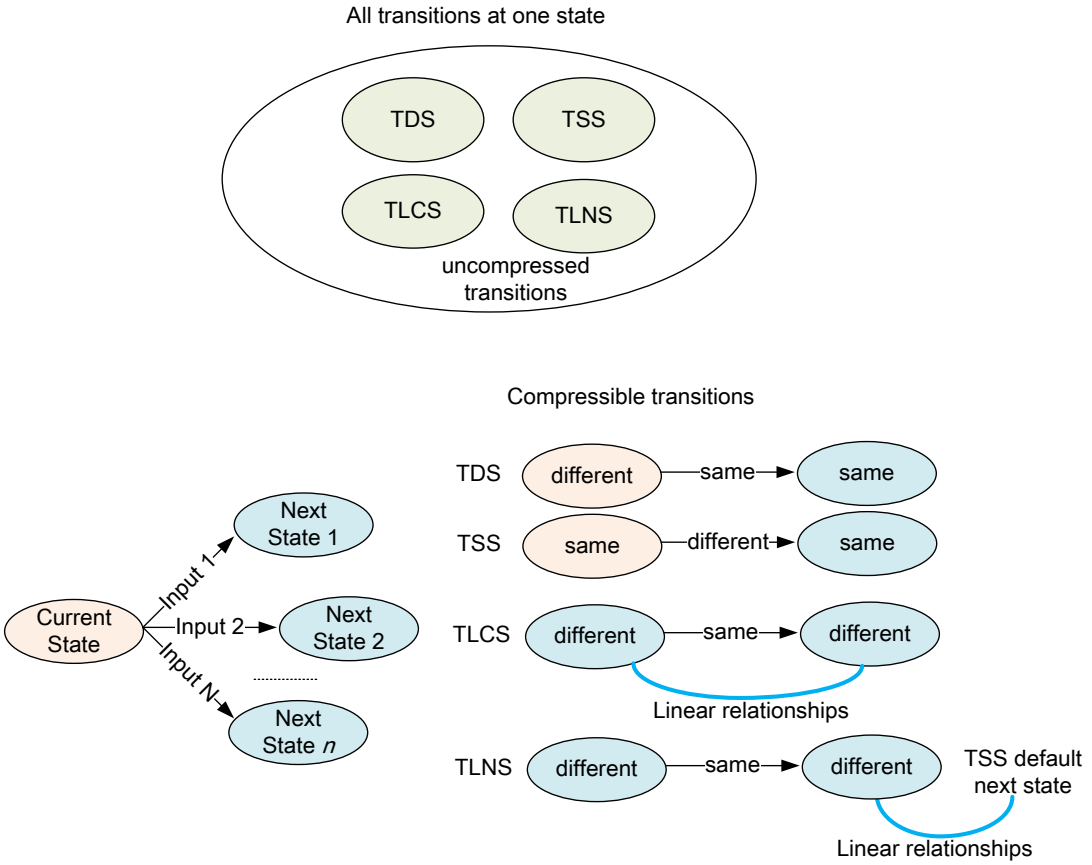


Figure 3-9 Groupings of compressible transitions in DFA

As an example from L7-filter, we choose state 3072 and 3074 in the DFA generated by "*imesh*" protocol, which is a media and file sharing client using P2P [110]. As shown in Figure 3-10, DFA turns to the next state after identifying the input character class. The compressible information will be classified into four groups based on DFA properties, through which they can be compressed separately.

State 3072:

Input Class	=>	Next State	Input Class	=>	Next State	Input Class	=>	Next State	Input Class	=>	Next State
0		***1505***	5		1506	10		1506	15		1506
1		1506	***6***		***1507***	***11***		***249***	**16**		**215**
2		1506	7		1506	12		1506	17		1506
3		1506	8		1506	**13**		**4**	***18***		***3101***
4		**8**	**9**		**6**	14		1506	19		1506
									20		1506

(a) state 3072 in DFA

State 3074:

Input Class	=>	Next State	Input Class	=>	Next State	Input Class	=>	Next State	Input Class	=>	Next State
0		***1724***	5		1725	10		1725	15		1725
1		1725	***6***		***1726***	***11***		***251***	**16**		**215**
2		1725	7		1725	12		1725	17		1725
3		1725	8		1725	**13**		**4**	***18***		***3103***
4		**8**	**9**		**6**	14		1725	19		1725
									20		1725

(b) state 3074 in DFA

Figure 3-10 Traditional DFA states for "*imesh*" protocol in L7-filter

According to the DFA table in Figure 3-10, groups are marked in different styles: bold underlined texts for Group 1 (e.g. States 4, 9, 13, 16), simple texts for Group 2 (e.g. States 1, 2, 3 etc.), shadowed texts for Group 3 (e.g. States 11, 18) and bold italic texts for Group 4 transitions (e.g. States 0, 6).

1) Group 1 of TDS

TDS stands for Transitions of Different states to the Same next state. In Figure 3-10, State 3072 and state 3074 have several similar transitions marked using bold underlined style, which have the same next hop states for same input characters and can be compressed. This redundancy is denoted by TDS. We also call this charateristic as inter-state similarity. TDS transitions can be compressed by using refering a TDS default transition, as shown in Figure 3-11.

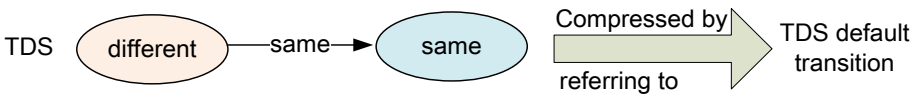
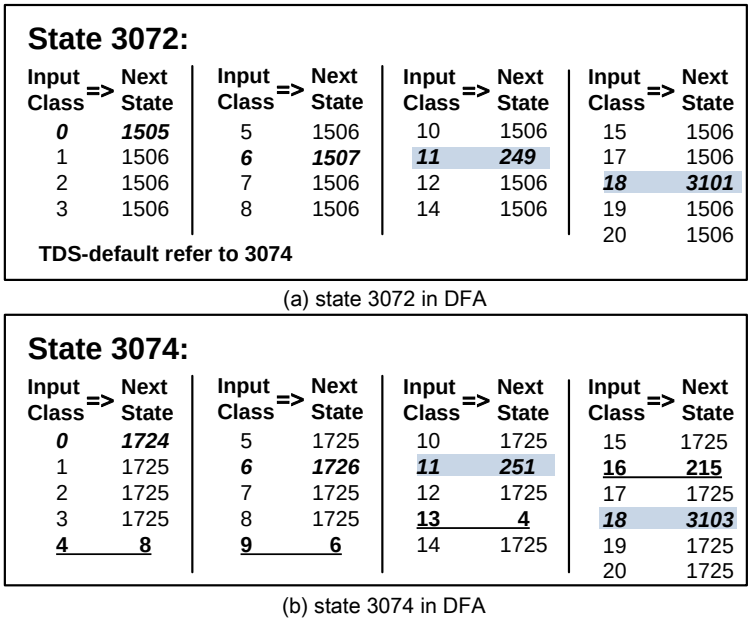


Figure 3-11 TDS compression

For example, TDS tranitions of state 3072 in Figure 3-10(a) can be replaced through a default transition and refer to those transitions of the same input classes in state 3074. The compressed transition table is shown in Figure 3-12. In this example,

there is only a small part of TDS transitions. Compared with Figure 3-10, 4 transitions can be eliminated and a default transition is added, then the number of transitions decreased by TDS compression is 3 in Figure 3-12.



State 3072:

Input Class =>	Next State	Input Class =>	Next State	Input Class =>	Next State	Input Class =>	Next State
0	***1505***	5	1506	10	1506	15	1506
1	1506	***6***	***1507***	***11***	***249***	17	1506
2	1506	7	1506	12	1506	***18***	***3101***
3	1506	8	1506	14	1506	19	1506
						20	1506

TDS-default refer to 3074

(a) state 3072 in DFA

State 3074:

Input Class =>	Next State	Input Class =>	Next State	Input Class =>	Next State	Input Class =>	Next State
0	***1724***	5	1725	10	1725	15	1725
1	1725	***6***	***1726***	***11***	***251***	**16**	**215**
2	1725	7	1725	12	1725	17	1725
3	1725	8	1725	**13**	**4**	***18***	***3103***
4	**8**	**9**	**6**	14	1725	19	1725
						20	1725

(b) state 3074 in DFA

Figure 3-12 Compression of TDS transitions

2) Group 2 of TSS

TSS stands for Transitions of the Same state to the Same next state. In Figure 3-10, a large part of transitions in one state, which are marked using simple texts, have the same next state. This redundancy is denoted by TSS. We also call this charateristic as intra-state similarity. Another default transition can be used for reduction, as shown in Figure 3-13. It should be noted that the two kinds of default transitions are different .

Figure 3-13 TSS compression

Accordingly, as shown in Figure 3-14 for state 3072, the 13 simple transitions to state 1506 can be omitted by referring to a TSS default transition. Similarly, 13 transitions of state 3074 can also be omitted by adding a TSS default transition to

state 1725. After TSS compression for this example, 26 transitions can be eliminated and 2 default transitions are added, then the number of TSS decreased transitions is 24. TSS and TDS default transitions are differentiated in the representation of DFA states.

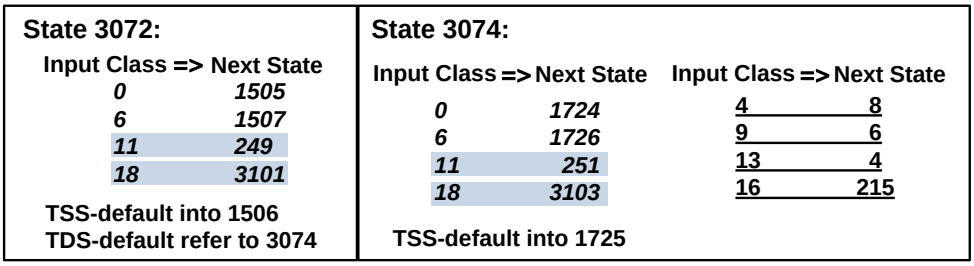


Figure 3-14 Compression of TDS and TSS transitions

3) Group 3 of TLCS

TLCS stands for Transitions whose next states have Linear relations with Current State ID. For shadowed transitions 11 and 18, although they point to different next states, there are some underlying connections between them. Their next states have linear relationship with current state, that is 3074-3072 = 251-249 = 3103-3101 = 2. The difference of current states is the same as that of next states for some transitions. This property can also be exploited for compression, by only storing the difference. And then TLCS can be transforme to TDS for compression, as shown in Figure 3-15.

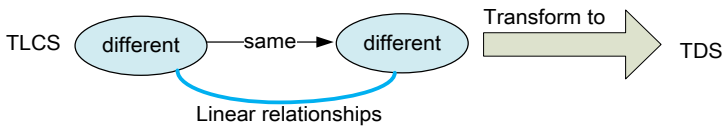


Figure 3-15 TLCS compression

In above example, based on this property, minus the current state number from its next state number, transitions 11 and 18 of state 3072 and state 3074 point to the same value, which is not a state number, as shown in Figure 3-16(a); thus they are similar to those in Group 2, and can be further compressed. Figure 3-16(b) illustrates the result processed with compression on three groups (i.e TDS, TSS and TLCS), where TLCS reduces 2 more transitions, but more information is necessary to label the transformation of Group 3 transitions.

State 3072:	State 3074:
Input Class => Next State	Input Class => Next State
11 249	11 251
18 3101	18 3103
Minus state ID=3072	Minus state ID=3074
<=> 11 -2823	<=> 11 -2823
18 -29	18 -29

(a) transformation of Group 3 transitions

State 3072:	State 3074:	
Input Class => Next State	Input Class => Next State	Input Class => Next State
0 1505	0 1724	4 8
6 1507	6 1726	9 6
		13 4
TSS-default into 1506		16 215
TDS-default refer to 3074	TSS-default into 1725	11 -2823
Minus state ID of input 11, 18	Minus state ID of input 11, 18	18 -29

(b) further compression using TDS after TLCS transformation

Figure 3-16 Transformation of Group 3 transitions for further compression

4) Group 4 of TLNS

TLNS stands for Transitions whose next states have Linear relations with the most popular Next State of current state. Some transitions have linear relations with the most popular next state, which is the TSS default transition, as shown in Figure 3-17.

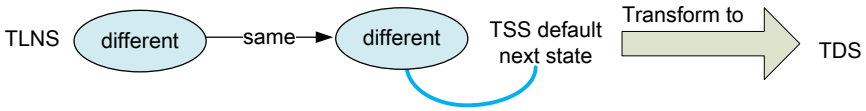


Figure 3-17 TLNS compression

In above example, the next states of transitions 0 and 6 have linear relations with their current most popular next state, that is 1505-1506=1724-1725 and 1507-1506=1726-1725, where 1506 and 1725 are TSS default transitions for state 3072 and state 3074 respectively. In this example, if we only preserve the difference, transitions 0 and 6 of different states will point to the same value, as shown in Figure 3-18(a); thus they are similar to those in Group 2 and can be further compressed. Figure 3-18(b) illustrates the result with the compression on four groups (i.e TDS, TSS, TLCS and TLNS), where TLNS reduces 2 more transitions. Similar to the compression on Group 3 transitions, extra information is required to record the

transformation on Group 4 transitions.

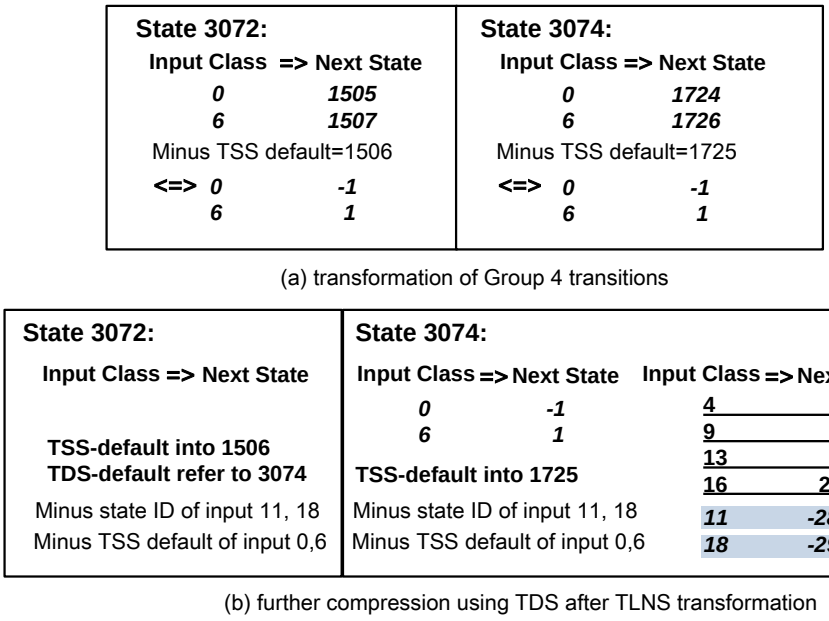


Figure 3-18 Transformation of Group 4 transitions for further compression

Next we show the matching process with the two states in Figure 3-18(b), where the next states for four groups of input characters are obtained respectively. If the current state is State 3074, four groups of input characters obtain their next states as follows: 1) with an input character such as "4", the next state is state 8; 2) with an input character such as "1", the next state is the TSS default state 1725; 3) with an input character such as "11", the next state is 251, as a sum of -2823 and state ID 3074; and 4) with an input character such as "0", the next state is 1724, as a sum of -1 and TSS default 1725. Similarly, if the current state is State 3072, four groups of input characters obtain their next states as follows: 1) with an input character such as "4", the next state is the TDS default state 3074, but the input character remains to be "4" for state 3074, and then it goes to state 8; 2) with an input character such as "1", the next state is the TSS default state 1506; 3) with an input character such as "11", the next state is the TDS default state 3074, and then it goes to state 249, as a sum of -2823 and state ID 3072; and 4) with an input character such as "0", the next state is the TDS default state 3074, and then it goes to state 1505, as a sum of -1 and TSS default 1506.

After compression with all four groups, the 42 transitions of the two states are represented by 8 transitions, with a reduction of 81%. On the other hand, whether or not to explore all kinds of redundancies depends the characteristics of system rule set. If only a small percent of transitions have TLCS or TLNS properties, the memory decrease is not worthwhile compared to the processing overhead.

3.4 Extend-D^{2}FA: Improved Algorithms on D^{2}FA

D^{2}FA consideres only one type of TDS redundancy. In view of the blind spots of D^{2}FA, we propose an extension method, called Extend-D^{2}FA, which can eliminate the first two types of TSS and TDS redundancies. The D^{2}FA method focuses on reducing transitions of different current states to the same next state (i.e. TDS). For further memory reduction, Extend-D^{2}FA also has capability of handling transitions from the same current states to the same next states. Extend-D^{2}FA keeps the advantages of D^{2}FA. And with a complementary kind of compression, Extend-D^{2}FA has a larger compression ratio when D^{2}FA can not effectively reduce the memory consumption of DFA transitions.

3.4.1 Algorithm Structure

In Extend-D^{2}FA, a base table is constructed to record the most popular next hop state (TSS state) for each state, as shown in Figure 3-19. In above example, the most popular next hop state for states 3072 and 3074 are 1506 and 1725 respectively in its base table. Next, we set corresponding transitions with TSS redundancy pointing to "-1". After transformation of state transitions, different states in DFA would have more transitions pointing to the same next hop value, which can be compressed using the same methodology of D^{2}FA.

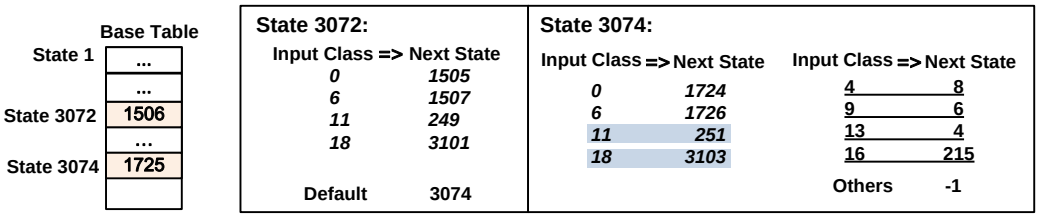


Figure 3-19 Based table and transitions compression by Extend-D^{2}FA

We compare the compression of Extend-D^{2}FA to D^{2}FA using this example. Following the discussion in Section 3.3.3 for the groups of TSS and TDS, with four TDS transitions removed and one new default transition added, three transitions can be eliminated by D^{2}FA between two states. After our modification, 26 TSS transitions can be removed by adding one TSS transition to -1 in state 3074, another 25 tranitions can be eliminated. This example achieves a total saving of 28 transitions. As this example shows, Extend-D^{2}FA can significantly improve compression ratio of D^{2}FA.

For the matching engine, there is a register storing the current state ID with next state set to -1 after reading a new byte character. While travelling across the default path, this value will not be changed. If the searching result is -1, next state refers to base table using register value as index. Otherwise, the next state is the searched result directly. For example, if the current state is 3072 and the input character belongs to class 4, it turns to state 3074 using default transition and find next state of 8. And if the input character belongs to class 1, it turns to state 3074 using default transition and find next state of -1, then it takes the value of 1506 in base table as the next state.

3.4.2 Experiment Results

The evaluation is based on the rule set of L7-filter [11], which contains 104 rules. The total transitions number of a table compressed DFA for the 104 rules is 791153. Table 3-10 shows the compression performance of D^{2}FA for L7-filter within certain number of default transitions. Due to its blind spots on certain rules, D^{2}FA cannot obtain a good compression ratio sometimes. Within the default path depth from 1 to 7, the compression ratio ranges from 45.2% to 53.2%. With our proposed scheme of Extend-D^{2}FA, the corrsponding compression ratio reaches from 80.3% to 93.8%.

Table 3-10 Compression comparison between D^2FA and Extend-D^2FA

Default path depth		1,2	3	4	5	6	7
D^2FA	transitions	433279	401909	383284	376467	372009	370199
	compression ratio	45.2%	49.1%	51.5%	52.4%	52.9%	53.2%
Extend -D^2FA	transitions	155411	110214	85132	71357	60357	48736
	compression ratio	80.3%	86.1%	89.2%	91.0%	92.3%	93.8%

*Total transitions of original DFA is 791153.

In comparison, the compression ratio of big DFAs is calculated. We choose the largest 20 DFAs out of the 104 rules in L7-filter. Suppose there are within 5 default paths in D^2FA and Extend-D^2FA, we compare the average compression ratios in Table 3-11 with 20 big DFAs. Extend-D^2FA also obtain a better compression ratio of 74.54% compared to 67.59% of D^2FA.

Table 3-11 Compression ratios of D^2FA and Extend-D^2FA with big DFAs

Methods	D^2FA	Extend-D^2FA
Comparison ratio	67.59%	75.54%

The rule sets from Snort and Bro NIDS are also considered. Table 3-12 shows the compression ratio of memory sizes of D^2FA and Extend-D^2FA within 5 default paths, and that of to the original table-compressed DFA. Based on Snort rule sets and Bro rule sets, the compression ratio could reach up to 99%. However, since the results for D^2FA are already very large (e.g. 98.4% compression rate on *Snort11* rule sets and 99.5% compression rate on *Bro648* rule sets), the improvement by using Extend-D^2FA is not obvious.

Table 3-12 Compression ratios with more datasets

Dataset	Comparison ratio with 5 default transitions	
	D^2FA	Extend-D^2FA
Snort11	98.4%	99.6%
Bro648	99.5%	99.8%

The creation cost of Extend-D^2FA is similar to that of D^2FA. Because the analysis of TSS redundancy is performed at the same time with TDS redundancy, Extend-D^2FA initialization and update cause negligible percent of extra time and memory compared to D^2FA. After the Extend-D^2FA is built, it only needs an

additional base table to store the TSS default transitions for all the states.

In summary, to be memory efficient, based on the analysis of different kinds of redundancies, we extend the classical D^2FA to Extend-D^2FA by introducing a base table. Extend-D^2FA can eliminate an extra kind of redundancy. Experiments show that Extend-D^2FA can achieve a larger compression ratio of transitions in DFAs with the same number of default transitions as D^2FA.

3.5 Tag-DFA algorithm

Based on D^2FA, another improved algorithm called Tag-DFA is proposed, which can obtain up to 90% compression ratio by the introduction of new states with tags to represent most of transitions in the original DFA states. For each input character, Tag-DFA can perform DFA lookup within two accesses of states in the worst case. In this way, a line-speed and steady throughput can also be guaranteed.

D^2FA has a significant drawback which restricts its utilization. For each byte to be matched, it would visit more than one state along the default path, which greatly limits its worst case throughput. Generally, there is a tradeoff between throughput and compression ratio. In this section, we propose a scheme called Tag-DFA to reduce more redundancies of traditional DFA. By introducing a few new states with tags, Tag-DFA can achieve a comparative high compression ratio while Tag-DFA generates little side effect on its throughput.

3.5.1 Algorithm overview

The above mentioned four kinds of redundancies in DFA summarized in Section 3.3 have never been considered together, and we intend to merge them into one scheme to increase the compression ratio. This scheme targets at reducing these redundancies as much as possible and solving the problem of multiple default paths in D^2FA.

Figure 3-20 shows Tag-DFA states with above example. A new state is introduced

to represent compressible information among multiple states. Meanwhile, tag is used to distinguish different kinds of redundancy in the new state and indicates how to get the next state within one hop. The base table records the most popular next hop state for each state.

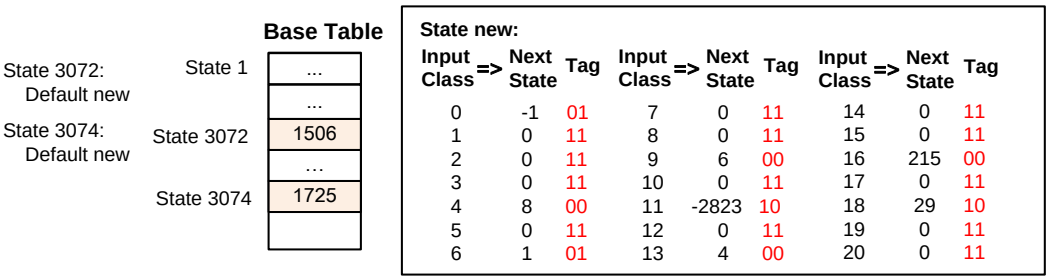


Figure 3-20 An example of Tag-DFA

The meanings of tag bits are shown in Table 3-13, each of which represents one of the four kinds of redundancies in Section 3.3. This example can be extended so that multiple states can be grouped together using one new state to store their common compressible information. The use of tags have two advantages: firstly, it gains a good compression ratio; second, the throughput of DFA matching in the worst case will have a fixed lower bound by introducing new states. It accesses at most two states for each byte in some conditions and only access one state in other conditions.

Table 3-13 Meaning of tags in Tag-DFA

Tag	Meanings
00	Next state is current value (TDS)
11	Next state is value in Base table (TSS)
01	Next state is sum of current value and current state value(TLCS)
10	Next state is sum of current value and value in base table (TLNS)

Overall, Tag-DFA maintains the basic architecture of DFA. A comparison between DFA and Tag-DFA is illustrated in Figure 3-21, where part of transitions of state *S* and state *M* defaultly refer to state *new*. The differences bewteen Tag-DFA and original DFA lie in three aspects: firstly, Tag-DFA maintains a base table, storing the most frequent next state of each state; second, in the newly introduced state, two bits tag is introduced to denote which kind of redundancy it belongs to; third, most transitions in traditional DFA can be replaced by the introduced new state with an additional default transition pointing to the new produced state.

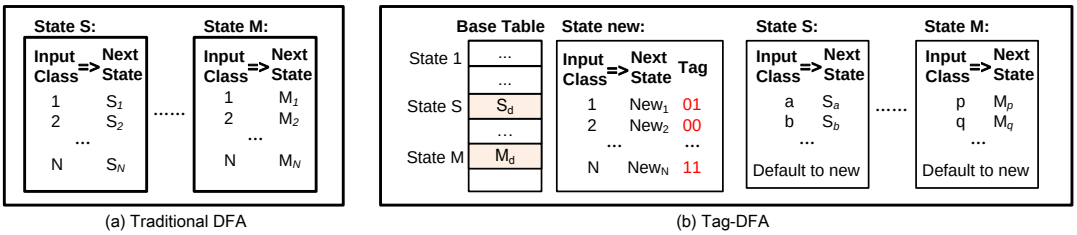


Figure 3-21 Comparison of DFA and Tag-DFA

3.5.2 Construction Procedure of Tag-DFA

Suppose that states with the same character class set have been grouped to produce a new state. All the states in DFA are partitioned according to its character classes. The construction of Tag-DFA includes the initialization of a new state with tags and the elimination of the redundancies from TSS, TDS, TLCS and TLNS, group by group. In particular, Tag-DFA is constructed in five steps: 1) initialize the new state; 2) eliminate redundancy of TSS; 3) eliminate redundancy of TDS; 4) eliminate redundancy of TLCS; and 5) eliminate redundancy of TLNS. And the compression ratio of a Tag-DFA, through the following construction steps, is highly dependent on the way of partitioning the state matrix.

Firstly, the new state is initialized with all transitions in the new state pointing to state 0, and the tag value for each transition is initialized to 11, which means TSS transition. A default transition to the new state is added for all the states in this group.

Secondly, for each state, the most popular next state is found and stored at its entry in base table. Then the transitions to the most popular next state are compressed from its state transitions, as shown in explanation of Group 2 (i.e. TSS).

Thirdly, for all states in a group, if there are transitions of different states to the same next state with the same transition label, they can also be compressed. Besides, the corresponding transitions in the new state are set to this next state value and their tag values are set to 00, as shown in explanation of Group 1 (i.e. TDS).

Fourthly, for certain input labels, if all of transitions under this input label in one group have linear relations with current state ID, these transitions can also be

compressed by setting their tag values to 01. The corresponding transitions in the new produced state point to a value equals to the difference of the values of next states and the value of current state, as shown in explanation of Group 3 (i.e. TLCS).

Finally, for an input label, if all the next states of transitions under this input label in one group have linear relations with the value in the base table, they can be eliminated by setting tag to 10. The corresponding transitions in the new produced state point to a value equals to the difference of the value of next state and the value of current most popular next state in the base table, as shown in explanation of Group 4 (i.e. TLNS).

3.5.3 Experiment Evaluation

We use L7-filter [11] rule set for the evaluation, which contains 104 rules. The total transitions number of the 104 rules is 791153. Table 3-14 shows the performance evaluation of Tag-DFA. It considers multiple redundant situations in design. Hence, it can obtain more than 90% compression ratio compared with traditional D^2FA algorithm. Meanwhile, the number of introduced new states is controlled to a low range, 99 and 106, which are smaller than 1% of total DFA states.

Table 3-14 Compression performance of Tag-DFA with different tags

Comparisons	DFA	Tag-DFA Tags =00 or 11	Variation Ratio	Tag-DFA Tags=00, 01, 10, 11	Variation Ratio
states	11499	11499+99 =11598	1% increase	11499+106 =11605	1% increase
transitions	791153	76618	90.3% reduction	59557	92.5% reduction

We also calculate the compression ratio of big DFAs for comparison. We choose the largest 20 DFAs out of the 104 rules in L7-filter [11]. Suppose there are within 5 default paths in D^2FA, we compare the average compression ratios in Table 3-15 with 20 biggest DFAs.

Table 3-15 Compression ratios of D^2FA and Tag-DFA with big DFAs

Methods	D^2FA: 5 default paths	Tag-DFA	
		Tag -00,11	Tag -00,01,10,11
Comparison ratio	67.59%	73.46%	78.57%

For the 20 largest DFAs in L7-filter, D^2FA and Tag-DFA can both obtain a good compression ratio. Especially for Tag-DFA, it gains more compression ratio than D^2FA with a default path length of 5, while the searching cost equals to D^2FA with default path length of 1.

Compared to D^2FA, the improvement of Tag-DFA is different depending on the circumstances. The compression of Tag-DFA covers that of D^2FA and Extend-D^2FA. But if the D^2FA has already obtained more than 98% of compression ratio, there is not much space for the improvement of Tag-DFA. On the other hand, it has a limited number of default transitions, which is at the cost of a larger construction complexity. All the experimental results were obtained on PCs with 1.20 GHz CPU and 1.93 GB memory.

Table 3-16 Time and memory cost of building D^2FA, Extend-D^2FA and Tag-DFA

Rule set	Compilation Time (*s*)			Memory Cost (MB)		
	D^2FA	Extend-D^2FA	Tag-DFA	D^2FA	Extend-D^2FA	Tag-DFA
L7-filter104	8.6	9.0	18.9	7.63	7.63	10.60
Snort11	20.1	20.3	20.4	6.31	6.32	7.30
Bro648	14.5	14.5	16.0	0.77	0.81	1.51

The running memory costs of the three DFA algorithms are around 256KB, which is smaller than the data cache of most modern processors (i.e. larger than 1MB). Next we compare the building time and memory cost of the three DFA algorithms, as shown in Table 3-16. Compared to the compilation cost of D^2FA (e.g. 8.6*s* for L7-filter), the generation of Extend-D^2FA does not require more compilation overhead (e.g. 9.0*s* for L7-filter), while that of Tag-DFA needs a longer compilation time (e.g. 14.9 for L7-filter). But for the rule set of *Snort11* and *Bro648*, there are not many other kinds of redundancies exept for TDS redundancy in their DFA transitions, thus their Tag-DFAs have the similar compilation time as that of D^2FAs.

3.6 Summary

DFAs are widely used to perform regular expression based pattern matching for network security. Most DFA compression algorithms consider only one type of redundant information in DFA. In order to be memory efficient, we analyze different kinds of inter-state and intra-state redundancies for further compression. Then we extend the classical D^2FA to Extend-D^2FA by introducing a base table, which can eliminate an extra kind of redundancy. Experiments show that Extend-D^2FA can achieve much larger compression ratio of transitions in DFAs with the same number of default transitions as D^2FA.

Based on D^2FA, an improved algorithm called Tag-DFA is further proposed, which can obtain a compression ratio of up to 90% by the introduction of new states with the tags to represent most of the transitions. For each input character, Tag-DFA can perform DFA lookup within two state accesses in the worst case so that a line-speed and steady throughput can also be guaranteed. Comparing Tag-DFA and Extend-D^2FA, Tag-DFA achieves a faster speed at the cost of a larger creation complexity; Extend-D^2FA can be more easily constructed as an extension from D^2FA, but similar to D^2FA, it may need multiple steps to process an input character. Because the two algorithms are different in construction, it is not possible to combine with the advantages of the two algorithms. The future work for Tag-DFA is to find an efficient method to partition DFA states into groups in order to maximize the compression ratio.

Chapter 4:

Extended Bloom Filter for Network Packet Matching

This chapter presents the use of multiple Bloom filters in parallel for hardware acceleration of DPI. Furthermore, an energy efficient adaptive Bloom filter model EABF is proposed for a balance of power efficiency and performance. According to traffic load, EABF algorithm, deployed on a two-stage or a multi-stage platform, adjusts the number of hash functions working in a Bloom filter. Thus the adaptive Bloom filter maintains a dynamic balance of power and processing speed. Moreover, for feasible and precise matching, the Bloom filter is implemented in combination with a cache mechanism and aimed at achieving a larger cache hit rate.

4.1 Hardware Acceleration of DPI with Bloom Filter

4.1.1 Motivation

With the expansion of the Internet, content inspection on network flows plays an increasingly important role in network security and network management. To enhance network security, NIDS uses DPI to match packet payloads against a set of previously extracted patterns to identify network protocols, applications or possible attacks. Currently, most of the software-based string matching algorithms cannot keep up with high-speed network, and hardware solutions are needed for DPI.

Since hardware platforms like FPGA have a limited memory resource, the compact structure of Bloom filters makes them suitable for DPI implementation on hardware platforms. Bloom filters use a set of hashing functions to map from a large

data set to a small and regular data set, and are widely used in network processing. Security patterns are mapped by Bloom filters to certain bits in a look up table. Pattern matching is simplified to checking specific bits in the look up table. However, traditional Bloom filters do not support pattern update and their matching results have false positives.

Accordingly, CBF with counters is used in this section to support the insertion and deletion of patterns. To eliminate false positives, an additional hash table is used to store the patterns mapped to each entry. The hash table with counters, which is frequently visited, is allocated on-chip and the hash table with patterns, which is seldom visited, is allocated off-chip. The elimination of false positives can be achieved by comparing with another table that stores the associated patterns of each entry in a look up table. Due to its large memory requirement, the new table cannot be held on-chip. Two optimizations of pruning and list balancing are applied to CBF, and significantly reduce the size of the hash table with patterns.

4.1.2 CBF for DPI design

DPI is implemented with Bloom filters on FPGA for hardware acceleration, as shown in Figure 4-1, where a pattern matching machine is constructed with a number of Bloom engines running in parallel. A Bloom engine consists of multiple Bloom filters. Each Bloom filter inspects a part of text of a unique length.

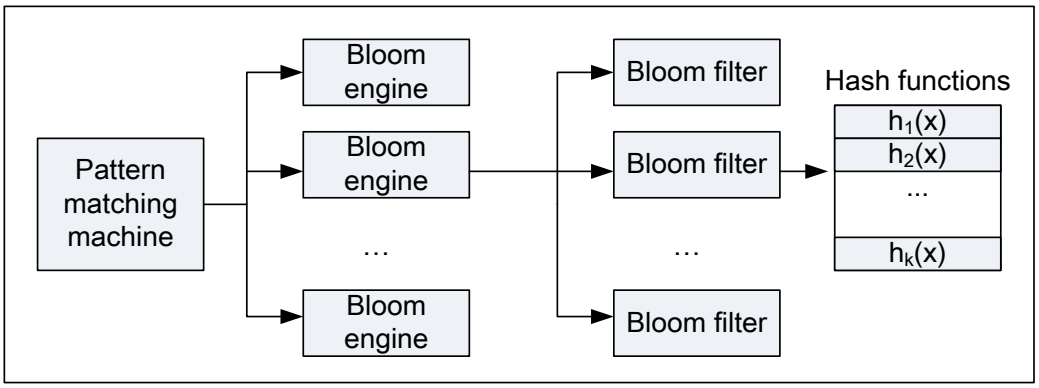


Figure 4-1 Structure of pattern matching with Bloom filter

CBF is used as the Bloom filter in Figure 4-1. By using counters instead of bits

in the hash table, called *T-counter* in experiment, CBF supports the insertion and deletion of items. To eliminate false positives, another table, called *T-pattern* in experiment, stores the mapped patterns corresponding to each entry. As a balance of memory and speed, *T-counter* is allocated on-chip and *T-pattern* is allocated off-chip. Slow access to off-chip memory makes false positive analysis after initial identification time-consuming. The analysis process can be improved with compact design and efficient access to *T-pattern*. For memory efficiency and smaller maintenance complexity, two kinds of optimized data structures, with ***pruning*** and ***list balancing*** introduced in [78], are applied to CBF hash tables, respectively.

● Pruned Table for CBF

The memory of *T-pattern* can be significantly compressed after pruning unnecessary copies of patterns in the table. Suppose a CBF has k hash functions. Among the multiple matched counters, false positive analysis can be performed by comparing with the pattern list with the minimum counter value. Suppose there are k hash functions, then as only one copy of a pattern is accessed during verification, the remaining (k-1) copies can be deleted.

The pruning procedure of removing redundant pattern copies is performed after the original pattern list table is built. For each item in set S, the pruning checks the k counters and deletes the (k-1) copies in corresponding positions except in the list of the smallest counter. Figure 4-2(a) is an example of a hash table with three patterns x, y, z, Figure 4-2(b) is the pruned result of Figure 4-2(a). When more than one counter has the smallest value, it holds the one that appeared first. In Figure 4-2, taking x as an example, when choosing the smallest counter, entries 1, 4 and 5 have the same value of 2. In this case, the system chooses the one with the lowest sequence number, and then x is stored in entry 1.

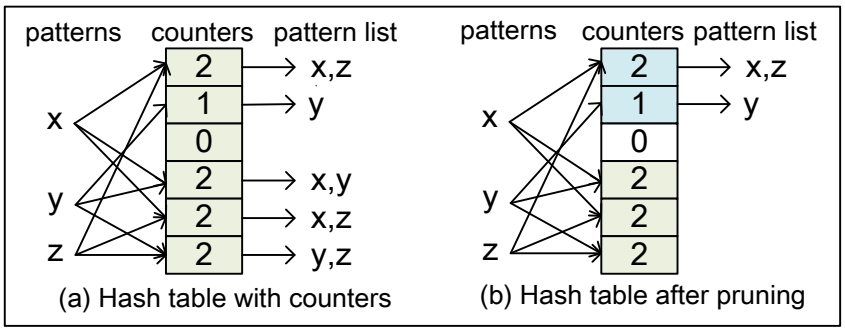


Figure 4-2 Basic and pruned hash table

On the other hand, the operation for insertion and deletion of a pattern becomes complicated as it also affects the mapping of other patterns. As shown in Figure 4-3, the insertion and deletion operations need to modify *k* corresponding counters, and they possibly change the minimum counter for some patterns and affect their stored positions in the pattern list table. Thus all the related patterns should be checked and re-inserted again. For instance in Figure 4-3(a), the insertion of a new pattern *w* increases the counters at entries 1, 4, 6, and it checks the minimum counter for the affected patterns *x*, *y*, *z*. In this case, counter in entry 5 becomes the minimum one for patterns *x* and *z*. Similarly, based on Figure 4-2(a), the deletion of an old element *z* decreases the counters at entries 1, 5, 6, and it checks the minimum counter for the related patterns *x*, *y*, as shown in Figure 4-3(b). In this case, their minimum counters are not changed.

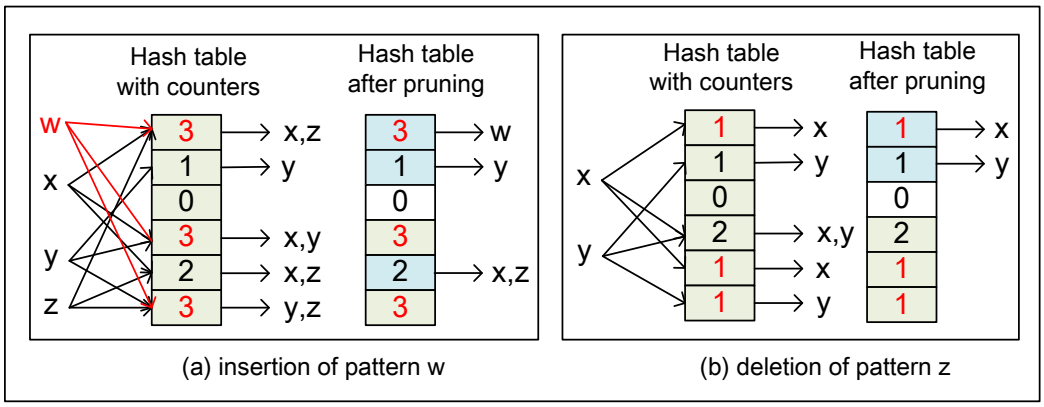


Figure 4-3 Pattern insertion and deletion on the pruned hash table

● List-balanced Table for CBF

Furthermore, the pruned hash table can be adjusted with list balancing optimization. Since the table list is allocated with fixed width, compressing the length of the longest list will reduce memory consumption. For example, pattern *x* and *z* are stored in the same entry in Figure 4-4(b), each entry in the pattern list should hold at least 2 patterns. Since Bloom filters only test if an entry is empty or not, an incremented value of a non-zero counter does not affect the matching result. As shown in Figure 4-4(c), *x* and *z* are stored at entry 1 with a counter of value 2. If the first counter is incremented to 3, the minimum counter positions of *x* and *z* change to entries 4 and 5 with counters of value 2, and patterns *x* and *z* can be moved to different entries. In this way, the maximum pattern list length is reduced from 2 to 1, which can save half the size of the pattern list with fixed memory allocation.

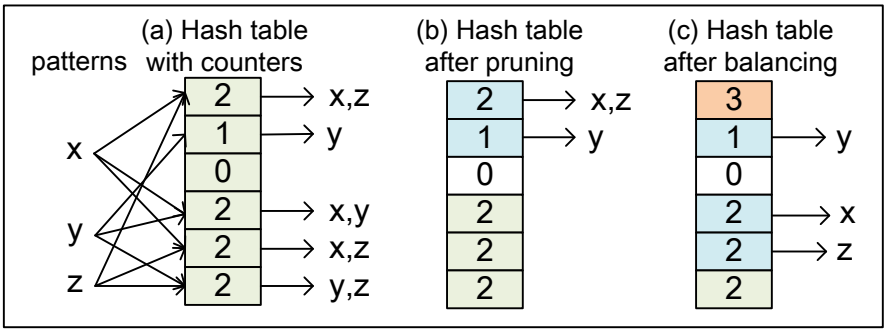


Figure 4-4 Basic, pruned and balanced hash table

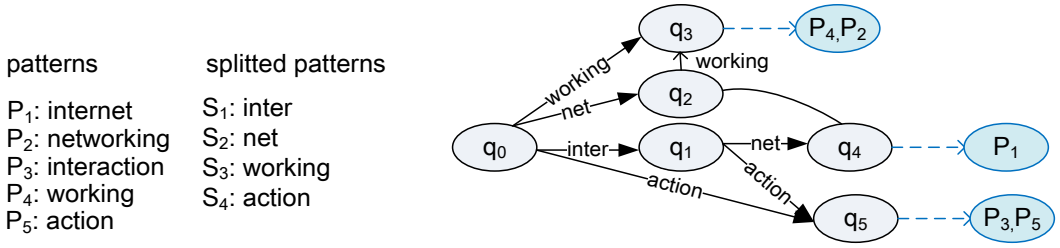
In particular, the list balancing process is performed by checking entries in the pruned hash table whose number of items is above a certain threshold (e.g. 3). Then the counters of these entries are incremented by one, and the related patterns in the pruned table are compared and inserted again. Moreover, CBFs allow the same pruning and balancing operations on dynamic sets that can be updated via insertions and deletions with larger memory requirements.

● Solutions for Very Long Patterns

For pattern matching of very long strings or strings with arbitrary lengths, Bloom filters can’t be used directly due to the memory limitation on hardware, but need to

work together with other types of pattern matching algorithms. Dharmapurikar et al. in [79] show that Bloom filter based pattern matching algorithms work well for patterns under 16 bytes (L_{max}<16) in most cases for network applications. However, considering the memory consumption and false positive rate, Bloom filter is not capable of directly handling very long patterns. In Snort rule set [9], although over 90% of patterns are within the limits, there are some very long patterns over 40 bytes in certain rules; considering the memory requirement, it is not practical to build a corresponding Bloom filter for these patterns.

The basic idea is to split up the long strings into multiple fixed lengths of short segments and use the bloom filter algorithm to match the individual segments. When the long pattern is generalized to multiple short strings, it can be transformed to a state automaton in which new symbols are formed by groups of characters. Figure 4-5(a) is the super alphabet consisting of multiple *k*-character symbols as a set of super characters to be matched by Bloom filters. Figure 4-5(b) is an NFA constructed with super characters in Figure 4-5 (a).



(a) Split long patterns into short patterns (b) Build an NFA using short patterns

Figure 4-5 Split long patterns into shorter strings for Bloom filter

The algorithm in [79] combines Bloom filters with the Aho-Corasic multi-pattern matching algorithm for arbitrary string length. A structure of Aho-Corasic NFA called Jump-ahead Aho-CorasicK NFA (JACK-NFA), jumps ahead by *n* characters each time, which is similar to the NFA in Figure 4-5(b). This scheme leverages the simple Bloom filter with Longest Prefix Matching (LPM) to perform segments' matching and tail matching. To match the string correctly, the JACK-NFA machine must capture it at the correct *n*-character boundary. However, a pattern cannot be detected if it

appears from the middle of an *n*-character string. To solve this problem, *n* machines are deployed in parallel each of which scans the text with one byte offset. To implement these virtual machines, it needs to maintain *n* independent states at the same time.

4.1.3 Hardware Architecture of Bloom Engine

- **Bloom filters for Pattern Matching**

The architecture of Bloom filters for DPI is shown in Figure 4-6, where pattern matching is performed using multiple Bloom filters in parallel. Based on pattern length, patterns are clustered into a number of groups. Each Bloom filter represents the detection for a group of patterns, and detects strings with a particular length ranging from L_{min} to L_{max}. As shown in Figure 4-6, multi-pattern matching with Bloom filters is performed in a text window of w bytes ($w=L_{max}$). Data in the text window is examined by k Bloom filters ($k=L_{max}-L_{min}+1$), and one byte can be moved ahead each clock cycle. In particular, XOR-based hash functions and polynomial hash functions are always utilized as hash functions in network processing. Here we use H_3 hash function [29] composed of a larger number of XOR operations, which are convenient for hardware design.

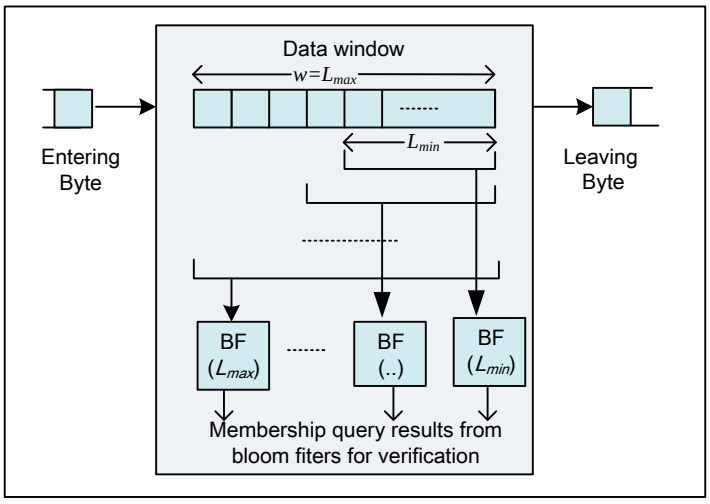


Figure 4-6 Pattern matching with Bloom filters of different lengths

● Multiple Bloom Engines in Parallel

In order to achieve higher throughput, multiple Bloom engines can be pipelined, each of which processes a *w*-byte window. With *G* Bloom engines running in parallel as shown in Figure 4-7, the system moves ahead *G* bytes each cycle, and the throughput is improved by *G* times.

Beginning at a particular position in the window, each Bloom engine checks all the possible occurrences of patterns within *G* bytes. There are (*w*+*G*) bytes in the window where each of the adjacent *w* bytes are allocated to each of the *G* Bloom engines. When more than one Bloom filter in a Bloom engine or more than one Bloom engine reports a successful match, the system need to report all the matched strings from the data window.

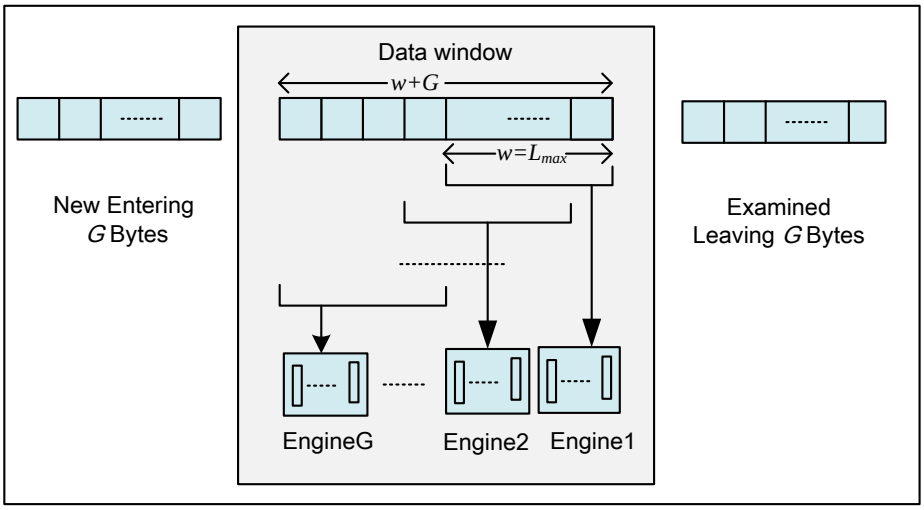


Figure 4-7 Multiple Bloom engines in parallel

This architecture is a common design and the hardware acceleration of DPI mainly depends on the parallel instantiation of multiple engines to improve the throughput. However, the hardware acceleration with multiple Bloom engines is at the cost of redundant design with larger memory consumption. In combination with memory compressed hash tables for Bloom filters, my architecture is a memory efficient of hardware acceleration architecture. Although the new structure significantly the memory consumption, it requires a longer initialization time. On the other hand, the matching time with hardware acceleration is maintained as that of the

basic CBF.

4.1.4 Experimental Evaluation

- **System Implementation**

There are three procedures in a pattern matching system: ***pattern matching***, ***pattern training*** and ***pattern updating***.

Firstly, the ***pattern matching*** procedure filters the input packets against the hash tables. Figure 4-8 shows the main blocks in a Bloom filter based DPI system, including the pattern matching block, verification block and statistic block. As shown in Figure 4-7, *G* Bloom engines are instantiated in parallel and the pattern matching block reads in *G* subsequent bytes each clock cycle. In order to reduce interface signals, the false positive analysis module is also included in a Bloom engine. Due to false positives in Bloom filters, the initially matched results are verified with the patterns in the pattern list table. Finally, statistical analysis on the matching probabilities and the false positive rates is performed on the final output matched results.

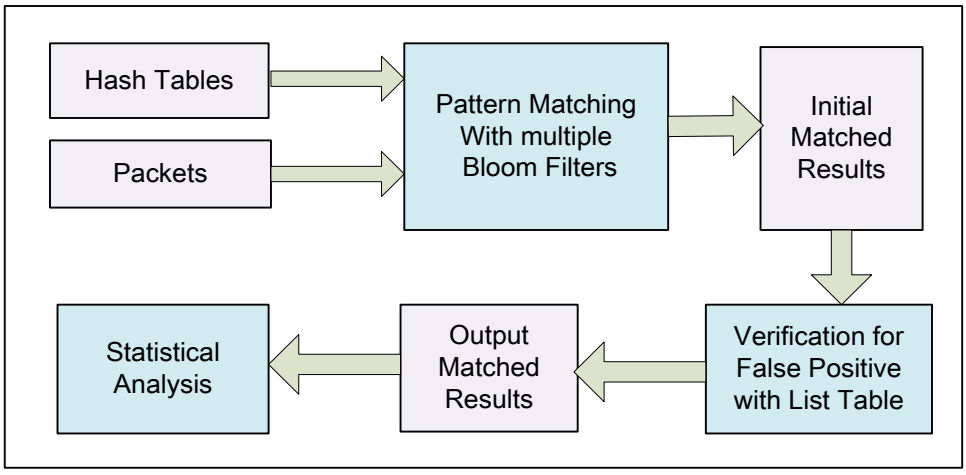


Figure 4-8 Pattern matching and statistical analysis flow

Secondly, ***pattern training*** generates (L_{max}-L_{min}+1) hash tables, each of which is related with a Bloom filter. All the bits are initialized to 0 and the mapped bits of

patterns are set to 1. For each pattern, if there is a pattern list table associated with the mapped bit, the pattern is appended to the tail of the list; or else, it generates a pattern list for this bit and puts the pattern in the list. When all the patterns are processed, the list table can be further pruned and balanced.

Figure 4-9 shows the ***pattern training*** procedure. There are *k* groups of patterns of length L_1, L_2, … , L_k, and each group has *n* patterns. The *k* Bloom filters, Bloomfilter_L_1 to Bloomfilter_L_k, produce two groups of *k* tables as *T_counter* and *T_pattern* for comparison in the pattern matching process. Based on basic hash tables, the ***pruning*** module generates the pruned hash table, *T_pattern_pruned*, by checking *k* hashed positions and deleting this pattern from all the pattern lists except for the one with the minimum counter. The ***verification*** block in Figure 4-8 only probes the pattern list with the smallest counter. And then the ***list-balancing*** module generates list-balanced hash tables *T_counter_balanced* and *T_pattern_balanced*. It examines the length of pattern lists with a threshold, and moves extra patterns in the lists over the threshold after matched positions with shorter pattern lists.

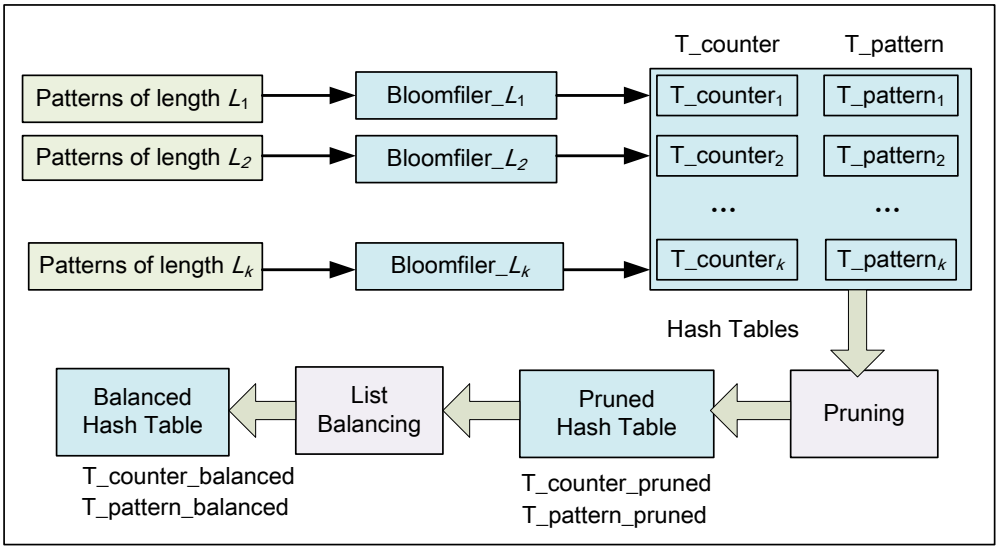


Figure 4-9 Building hash tables for pattern training

Thirdly, ***pattern update*** inserts new patterns and sets the corresponding bits in the hash tables and the pattern tables, similar to that of the ***pattern training***.

As shown in Figure 4-8, the hash tables *T_counter* and *T_pattern* generated in

Figure 4-9 are used as input to the multi-pattern matching system. In the pattern matching process, if all the mapped entries in $T\_counter_i$ (i=1..k) are above 0, the pattern matching module compares the text in the window with all stored patterns in $T\_pattern_i$ (i=1..k). The long patterns can be represented with numbers for simplicity, in experiments, the patterns have a sequence in the set from 1 to n, the new pattern is assigned as (n+1); then in order to save space in the *T_pattern* hash table, the backup of basic CBF hash table can keep the sequence numbers instead of the full patterns.

● System Evaluation

The pattern matching system is implemented in Verilog HDL and simulated in ModelSim. After that, the system design is synthesized in Quartus II based on the Altera Stratix III series FPGA to obtain resource consumption data.

For system evaluation, the patterns used in experiment are extracted from the Snort rule set [9] with specific lengths from 2 bytes to 16 bytes (that is L_{min}=2 and L_{max}=16). As a comparison, Bloom filters are designed with 4 or 8 hash functions respectively. In experiment, there is no overflow for the smallest counter of each pattern within 16 bytes. Experiment shows that no more than two patterns fall in the same entry in the table *T_pattern_pruned* after pruning. Furthermore after balancing, each entry has no more than one pattern in the table *T_pattern_balanced*.

Table 4-1 shows the memory consumptions of basic, pruned and balanced hash table. The entry of hash table with counter *T_counter* is four bits wide and the width of table *T_pattern* depends on its related pattern width. It is shown that pruned hash tables greatly reduce memory consumption by more than 20%. Furthermore, list balanced hash tables reduce by about 30% memory consumption.

Table 4-1 Memory sizes with different CBFs

Bloom Engines	Hash Functions	Memory Consumption of Bits				
		Basic	Pruned	Reduction	Balanced	Reduction
2	4	38425	27366	28.8%	24902	35.2%
2	8	45327	31890	29.6%	28042	38.1%
4	4	84562	65260	22.8%	58930	30.3%
4	8	109367	86278	21.1%	80273	26.6%

It is noted that the memory reductions are achieved at the cost of the longer initialization time. Based on the original tables of CBF is built, it needs extra time for the pruning and balancing procedure to generate the pruned hash tables (i.e. *T_counter_pruned* and *T_pattern_pruned*) and the balanced hash table (i.e. *T_counter_balanced* and *T_pattern_balanced*). After the initialization stage, the matching with pruned and list-balanced CBF requires nearly the same number of clock cycles as that of the basic CBF. In the experiment, the training time of pruned CBF and list balanced CBF are 11.3% and 14.3% more than that of the basic CBF, respectively.

4.2 EABF: Energy Efficient Adaptive Bloom Filter

4.2.1 Motivation

In recent years, the statistics of network energy consumption reported by ISPs show an alarming and growing trend. Internet power consumption is estimated to be over 4% of the whole electricity consumption as the access rate increases [80]. To support the fast growth of customer population and the expanding offer of new services, ISPs need a large number of high speed network devices to process the huge volume of traffic data. The drive for low power comes from environmental and economical motivations.

Therefore, for the emerging future Internet, it is recognized that energy efficiency should become part of the network design criteria, besides other basic concepts and key aspects [81]. Network applications primarily rely on hardware platforms to keep up with network speed. However, silicon technologies improve their

energy efficiency at a slower pace, namely, by a factor of 1.65 every 18 months, whereas router capacities increase by a factor of 2.5 every 18 months [82]. Thus it is difficult to reduce power consumption while guaranteeing network processing capacity based on current technology. In fact, power efficient Internet is still at an exploratory stage, while the energy aware key components in new router architectures are the promising points to work on.

Efficient Bloom filter design will reduce power consumption for various network applications. S. Dharmapurikar et al. in [79] use parallel Bloom filters to accelerate the pattern matching of DPI. Currently there is not much research on Bloom filter aimed at power efficiency. Existing Bloom filter based low power solutions are achieved at the cost of performance degradation, such as the fixed two-stage scheme in [83] or the fully pipelined multi-stage scheme in [84]. Compared to the regular Bloom filter approach, although the fixed stage schemes in [83] and [84] show significant power savings for average traffic conditions, they cannot handle peak traffic and seriously affects processing speed in the worst case. E. Safi et al. in [85] propose a low power architecture at VLSI level, but the design is specialized and cannot flexibly adapt to new scenarios.

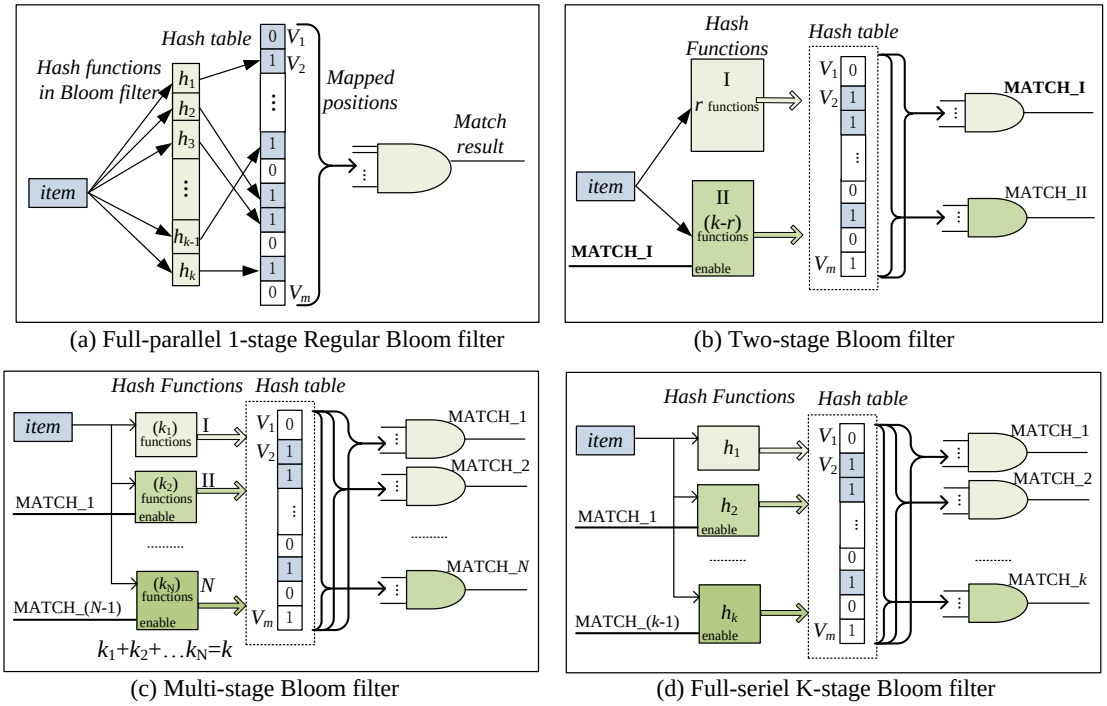
Our objective is to reduce overall energy consumption of Bloom filter based network applications without adversely affecting the processing latency. A Bloom filter uses k independent hash functions to map each input string to an array of m bits, which is initially trained with n patterns. The input string is considered as a pattern if all the mapped entries are equal to 1. It can be safely filtered if at least one of the entries is not 1. However, the matched results include false positives and the False Positive Rate (FPR) f is shown in Equation 4-1, where m is the length of hash table, n is the number of patterns and k is the number of hash functions.

$$f \approx \left(1 - e^{-kn/m}\right)^k \tag{4-1}$$

The parameter k that minimizes f is $k_{opt} = (m/n)\ ln2$ [49]. Fortunately, the short lookup time and small memory outweigh this drawback for many applications of

Bloom filters.

As shown in Figure 4-10, the k hash functions can be deployed in more than one stage for power efficiency. Figure 4-10 (a) is a regular Bloom filter with k hash functions, and the match result depends on the k mapped positions from the hash table. Figure 4-10 (b) presents a 2-stage Bloom filter [83] for low power design in DPI, where Stage I has r hash functions and Stage II has (k-r) hash functions. As shown in Figure 4-10 (c), the hash functions can also be divided into multiple groups. Stage I is always active and Stage II is activated only when Stage I produces a match. Part of hash functions are put in the second stage, which contributes to power savings. But the latency is increased for a match. Furthermore, M. Paynter in [84] uses a fully pipelined Bloom filter architecture with each function per stage, as shown in Figure 4-10 (d). However, it is at a much higher cost of latency. Simulation shows that the average latency might be ten times longer than that of a regular Bloom filter. In the worst case, its query latency could be k times longer than that of a regular Bloom filter.



(a) Full-parallel 1-stage Regular Bloom filter

(b) Two-stage Bloom filter

(c) Multi-stage Bloom filter

(d) Full-seriel K-stage Bloom filter

Figure 4-10 Structure of regular and multi-stage Bloom filters

In this section, an energy efficient adaptive Bloom filter, EABF, is proposed as

an optimal balance of power and latency. EABF is capable of adjusting the number of active hash functions according to the current workload automatically; in other words, *r* in Figure 4-10 (b) is a dynamic value. Hash functions in EABF can be switched among three states: working in the first stage, sleep or working in the second stage. The first stage is always active, whereas the second stage is activated by a match output from the first stage.

The subsequent contents in this section will present: 1) a adaptive two-stage platform for EABF; 2) three control strategies for the movement of hash functions between the two stages to adapt workload; 3) a key control circuit aimed at changing a hash function's working stage flexibly in one clock cycle and reducing both dynamic and static power consumptions. Experiments show that EABF can significantly reduce power so that it is similar to that of a fixed stage low power Bloom filters [83][84]; moreover, it decreases their long latency to be nearly 1 clock cycle as in a regular Bloom filter.

4.2.2 Two-stage Adaptive Bloom filter

Networks over-provide link bandwidth and other processing capacities in order to handle busy-hour traffic, which leads to a low average link usage of fewer than 30%. It is a huge waste of energy that network applications still consume energy even for idle periods. On the other hand, the network is designed to guarantee busy-hour performance.

The adaptive EABF is proposed on the two-stage architecture. The first stage is always active and the second stage only works when the first stage produces a match, similar to the scheme in [83]. EABF has the capability to control the active number of hash functions according to the work load automatically. Figure 4-11 shows an example of *k* hash functions, H_1 to H_k. At least *min* hash functions are maintained in Stage I for a tolerable false positive rate; according to incoming workload, the other (*k-min*) hash functions can be moved out to Stage II for low power or moved back to Stage I for faster response.

Stage II initially sleeps and is enabled by a positive MATCH_I, therefore, Stage II is activated if and only if it has hash functions and Stage I produces a match. In other words, if the text is matched by *r* hash functions in Stage I, it will be further processed by (*k*-*r*) hash functions in Stage II next clock cycle. The hash table is shared by all the hash functions. The final match could be MATCH_I, or MATCH_II if Stage II is enabled.

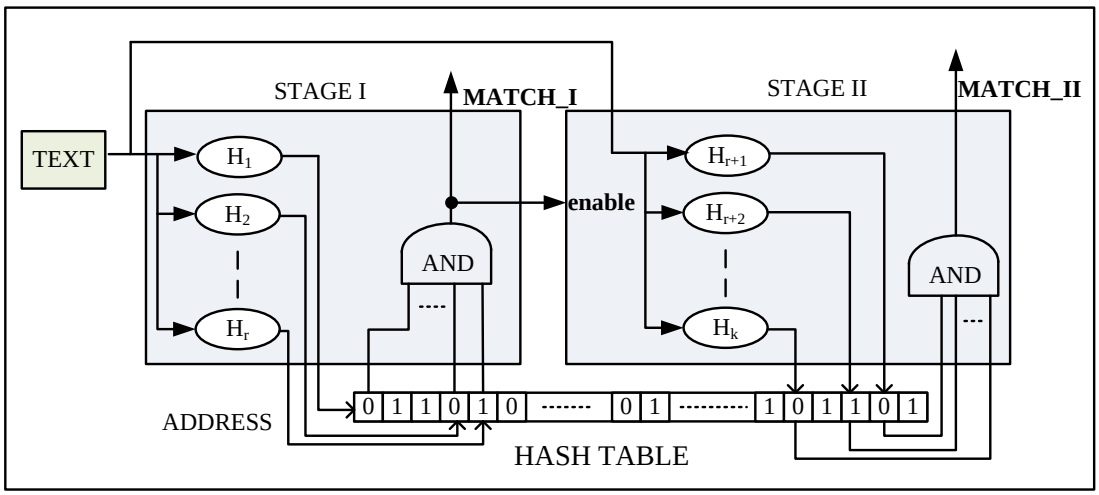


Figure 4-11 Two-stage Bloom filter

As shown in Figure 4-12, there are three possible states for a hash function in each clock cycle: active in Stage I, active in Stage II or sleep. Each hash function is controlled by a control bit C[*i*], value 1 means H_i is in Stage I and value 0 means H_i is in Stage II. Since only a non-matching leads to C[*i*] = 0, HF_i goes to Sleep state when it moves from Stage I to Stage II. If Stage I produces a positive match (i.e. MATCH_I = 1), HF_i works in Stage II. When C[*i*] = 1, it is brought back to Stage I.

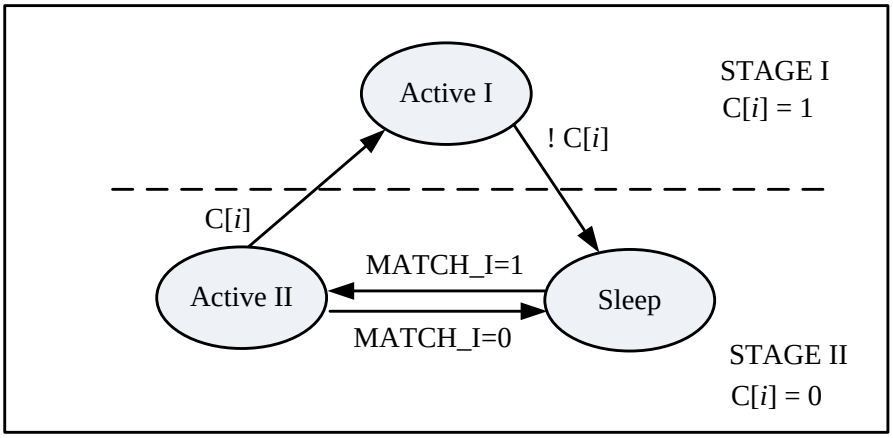


Figure 4-12 States of hash function *i*

Essentially, a hash function adapts its state by observing the Bloom filter input text and the match results. An invalid input means there is no effective traffic load. This corresponds to low traffic conditions or packet intervals. Moreover, when there is continuous data input without a match for a long time, it probably belongs to normal traffic. Since a Bloom filter has no false negative, less hash functions can handle the packet inspections. Then we can gradually move some hash functions to the sleep state in Stage II.

A match output means a pattern is recognized. Since malicious attacks like viruses appear frequently during certain periods in network applications, hash functions need to be brought back to Stage I quickly after a match to check subsequent packets.

Compared with the fixed two-stage scheme in [83], EABF provides different treatments for different traffic conditions. Depending on control policy, it adapts to workloads so that its average latency is shorter than that of the fixed two-stage scheme [83]. On the other hand, the power saving of the fixed two-stage scheme [83] is restricted by the number of hash functions in the first stage, which cannot be too small to maintain acceptable false positive rate. Due to its flexible adaption, EABF does not need to worry about the worst case latency in pursuit of maximizing power savings. In this way, EABF could be configured with less hash functions than that of the fixed schemes in [83], which leads to larger power savings. In conclusion, for low

link usage or clean traffic, the adaptive scheme acts like the fixed schemes in [83] for maximum power savings. For busy-hour traffic, it returns to a regular Bloom filter for fast processing.

4.2.3 Adaption Control Policy

The improvement of EABF is achieved by hash function adaption between Stage I and Stage II. The use of this adaption control policy largely determines system response speed and the percentage of power savings.

There are two special states in a two-stage EABF architecture: performance optimization state S_1, which is defined as a state that all the hash functions work in Stage I; and power optimization state S_2, which is defined as a state that only a minimum number of reserved hash functions work Stage I. The other states are different performance and power trade-off states.

In order to prioritize performance, rather than equally treating function movement, the ***easy move-in and hard move-out*** principle is used, so that Stage I hash functions are slowly moved out to Stage II and quickly moved back to Stage I. ***Move-in*** means more hash functions in Stage I, which leads to a smaller power saving ratio and a smaller delay. Contrarily, ***move-out*** means less hash functions in Stage I, which leads to a larger power saving ratio and a longer delay. The following three control strategies are presented for comparison of latency and power savings.

- **Policy A: Slow Hash Function Adaption**

A hash function is moved to Stage II when there is an invalid input or when there is no match for *T* clock cycles, and moved back to Stage I when there is a match. A timer is needed for keeping the number of filtered packets, which means a period of clean traffic. Figure 4-13 illustrates bit vector *C* for the control of (*k-min*) hash functions.

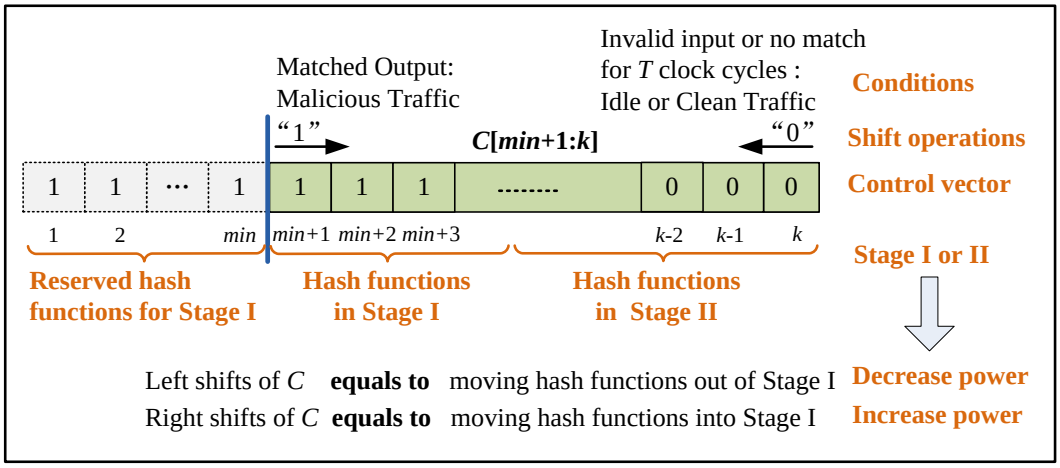


Figure 4-13 Control Vector Value

Initially, the Bloom filter can start from S_1 state where all the bits in C are equal to "1". Or it can start from the state that part of the total hash functions are in Stage I on the condition that $C[1:min] \equiv 1$ (that is, at least *min* hash functions are reserved in Stage I). For example, half of the total hash functions in Stage I and the others in Stage II, $C\left[min+1:\lceil k/2 \rceil\right] = 1$ (set to 1) and $C\left[\lceil k/2 \rceil+1:k\right] = 0$ (set to 0).

Functions with the larger indices are moved preferentially to Stage II. Conversely, functions with the smaller indices are moved preferentially back to Stage I. Correspondingly, as shown in Figure 4-13, the left shift of the control vector adds zero to the last bit (i.e. moving out a hash function), and the right shift of the control vector adds one to the front bit (i.e. moving in a hash function).

- **Policy B: Fast Hash Function Adaption**

More than one hash functions could be moved each time for faster response. For example, when a match happens, all Stage II functions can be moved back to Stage I simultaneously. Likewise for adaption to an idle network, when it detects invalid inputs or when there has been non-match for a number of clock cycles, it could move all the movable hash functions to Stage II.

Although this policy adapts faster, it might cause frequent adaption of hash functions, which leads to longer delay and affects performance. Counters can be used

for smoother adaption, where the left and right shift also refers to a corresponding counter value. The ***left-counter*** is increased by one for one assertion of ***left shift*** condition and reset to 0 for one assertion of ***right shift*** condition; the ***left shift*** happens when its counter reaches the threshold value and is reset the counter. It applies similarly for ***right-counter***. Moreover, the ***left-counter*** should be larger than the ***right-counter***, following the ***easy move-in and hard move-out*** principle for performance priority.

- **Policy C: Group Hash Function Adaption with Counters**

The decrease of hash functions affects the false positive rate (f) in a non-linear way. According to Equation 4-1, suppose n=1K, m=32K, $k_{opt}=(m/n)\ ln2$=22, and so when k changes from 1 to k_{opt}, f changes differently.

Table 4-2 Group of hash functions

***k* ranges**	**1:4**	**5:7**	**8:11**	**12:22**
f	$f>1.9\times10^{-4}$	$f<5\times10^{-5}$	$f<4.5\times10^{-6}$	$f<6\times10^{-7}$
Groups	I: i=1:4	II: i=5:7	III: i=8:11	IV: i=12:22

From Table 4-2, f changes very slightly when k ranges from 12 to 22, these values can be regarded as a group, and the 22 hash functions are divided into four groups. As Figure 4-14 shows, the control vector C is accordingly divided into three blocks for Group II, III and IV. Group I is reserved for Stage I and its hash functions cannot be moved to Stage II in principle.

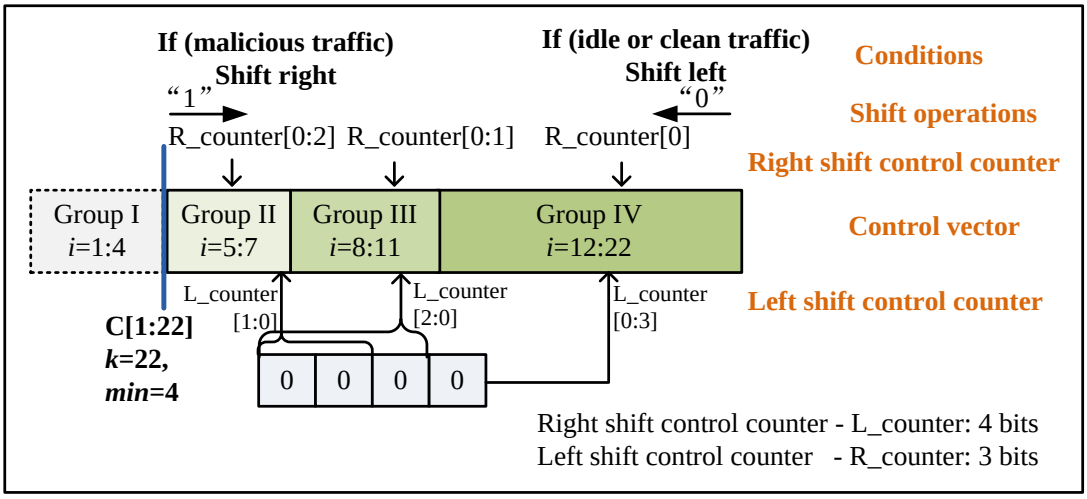


Figure 4-14 Control vector with left and right counters

The group-adaption Policy C with different moving speed can work together with Policy A or Policy B. A four-bit *L_counter* restricts left moving speed and a three-bit *R_counter* restricts right moving speed. If the algorithm is working with Policy B, a block is operated by one move instruction. If the algorithm is working with Policy A, Group II hash functions can be moved out of Stage I if there are 3 continuous invalid inputs; similarly, 7 for Group III and 15 for Group IV. For right shift, it needs 7 matches for Group II, 3 for Group III and only 1 for Group IV, for a hash function or a group of hash functions to move back to Stage I. The difference of restrictions is also in accordance with the ***easy move-in and hard move-out*** principle.

4.2.4 Key Component Design in Hardware Implementation

The adaptive Bloom filter can be implemented using software or hardware solutions. A hardware design that can be used in high speed networks is shown here. A key component for EABF is how to control a hash function HF_i to work in Stage I, Stage II, or sleep. The control circuit in Figure 4-15 is able to move HF_i to Stage II and return it to Stage I.

The left part of Figure 4-15 is the control circuit for hash block input signals including data, clock, power supply and reset, where CLK is responsible for dynamic power reduction and VCC is responsible for static power reduction. The right part shows how the output address in the hash table works for the match results. It is assumed that the power gating technique can disable parts of circuits by cutting off the power supply VCC to parts of circuits.

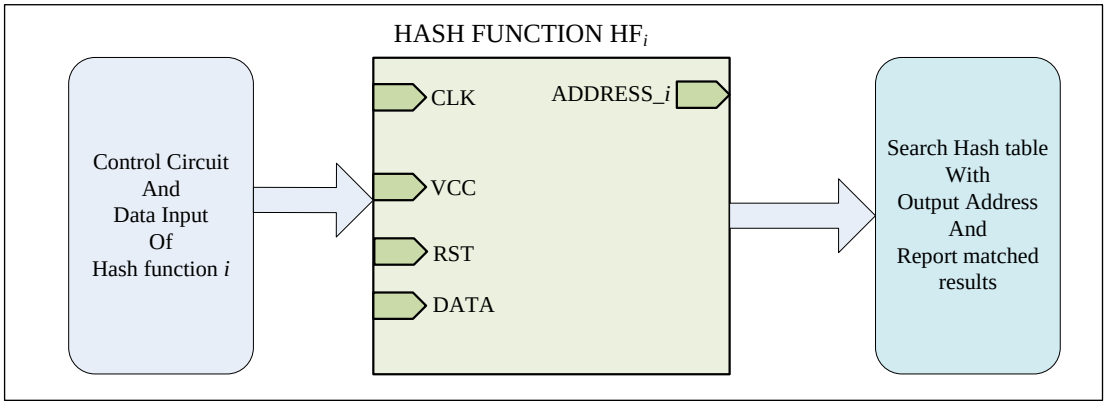


Figure 4-15 Control circuit of Hash function *i*

Figure 4-16 shows the control circuit of CLK, VCC and RST signals for the block of the hash function HF_i. CTRL_*i* is determined by the control bit C[*i*] and the matching result of Stage I. When CTRL_*i* is equal to zero (i.e. HF_i is in Stage II and the text does not match Stage I), HF_i sleeps and its functional block is reset with clock and power supply cut off. When CTRL_*i* is equal to one, HF_i works with normal clock and power supply.

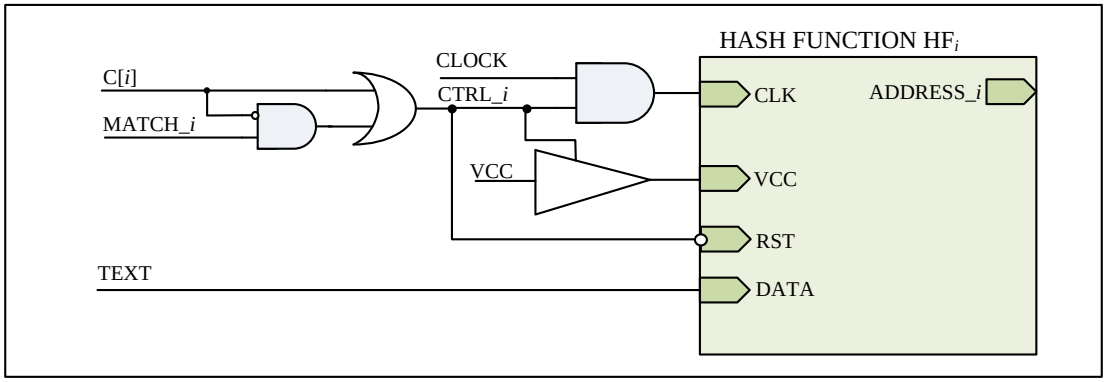


Figure 4-16 Control and data input signal

Figure 4-17 shows the control circuit of output result and MATCH. The difference between working in Stage I and Stage II is the effectiveness of the match result. If a hash function HF_i works in Stage II at present, its *C*[*i*] equals 0 and the value after OR gate equals 1, then the output of hash result HF_*i* will not affect MATCH_I. The value of MATCH_II relates to the results of all the hash functions in Stage II, and MATCH_II is zero if one of the hash results HF_*i* is zero.

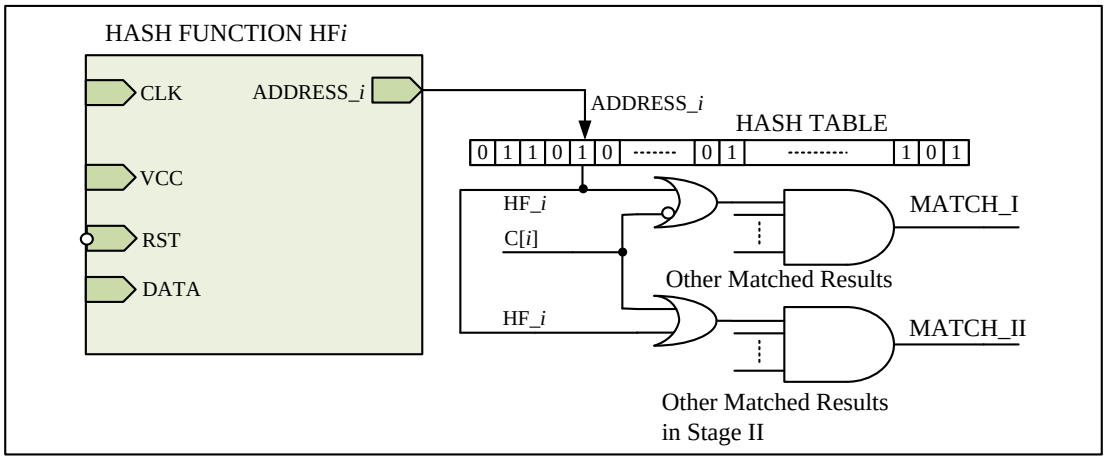


Figure 4-17 Address and output match signal

Figure 4-18 shows an example of a control circuit for match result, where an EABF has 4 hash functions: $HF_{1\text{-}2}$ in Stage I and $HF_{3\text{-}4}$ in Stage II, and the two stages are both in working state. The four bit control vector *C*[1:4]=1100. The input text is matched by HF_1 and HF_2 in Stage I and passed to Stage II, where HF_3 matches but HF_4 does not match. From the truth table, HF_4 works for Stage II output, but does not affect the matching of Stage I.

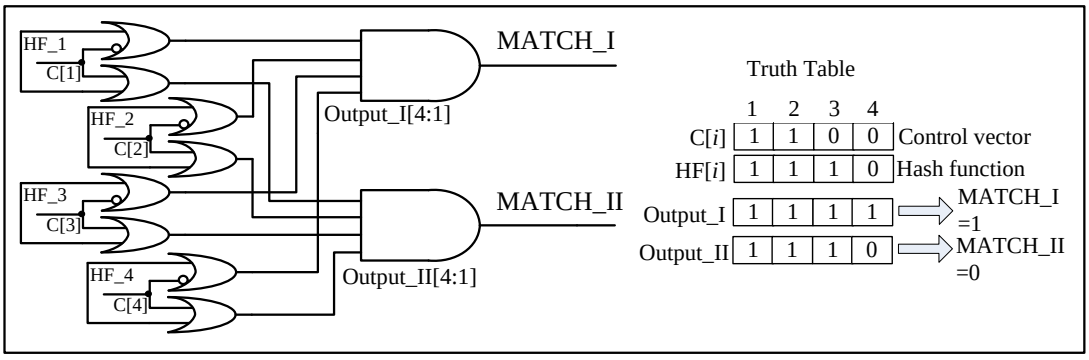


Figure 4-18 An example of control circuit for EABF

4.2.5 Performance Analysis and Experiment Results

The EABF has not been implemented on FPGA currently and the hardware implementation is my future work. The experiment in this section is based on software simulation.

Suppose each hash function maps evenly to *m* entries in the hash table, the match

probability (p) with random text for one hash function is shown in Equation 4-2. Considering the random input, it is assumed that a hash function produces an equal chance for "0" or "1" in general condition; therefore, p = 0.5.

$$f = 1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-kn/m} = p \tag{4-2}$$

True Positive Rate (TPR) is considered in evaluation, which is the true positive matching probability generated with the input traffic. Since the majority of network traffic is benign, the alert rate of TPR is about 0.1% [83]. Match probability p of a single hash function increases consequently, e.g. when k=22, p^{22}=TPR=0.001, the match probability of a single hash function is p= 0.7305.

● Analysis of Delay and Power Savings for a Two-stage Platform

Suppose EABF has r hash functions in Stage I and the other (k-r) hash functions in Stage II. For simpler analysis here and in order to be in consistent with the assumptions in [83], the second stage delay is assumed to be the same as the first stage delay of τ, τ is the clock period. The delay time T= τ when r=k; and when r<k, the delay time is shown in Equation 4-3.

$$T_{2\text{-}max}=2\tau, \text{ and } T_{2\text{-}avg}= \tau + p^r \cdot \tau \tag{4-3}$$

Power Saving Ratio (PSR) is an important measurement for the low power Bloom filter design. The PSR is defined as the reduction of power consumption between the basic and the new methods divided by the power consumption of the basic method. For Bloom filter, the PSR can be calculated as 1 minus the ratio of the number of active hash functions and the total number of hash functions. PSR depends on the number of active hash functions, and the power consumption of a single hash function does not affect the PSR. Another measurement of the traffic is the Link Clean Ratio (LCR), defined as the percentage of clean packets among all packets. In this experiment, clean packets lead to negative match and safely pass the Bloom filter.

The PSR of EABF in Equation 4-4 is the same as that of [83], except that r is a dynamic value here. The PSR of EABF could be a little larger since average r may be

smaller than the first stage hash function number of the fixed two-stage scheme in [83]. However, it has smaller latency than the fixed two-stage scheme in [83] since it can return to the regular Bloom filter.

$$PSR = \left(1 - \frac{r + (k-r)\,p^{r}}{k}\right) \cdot 100\% \qquad (4\text{-}4)$$

It is known that *r* changes between *k* and *min*, where *k* is the total number of hash functions and *min* is the minimum number of hash functions in the first stage. The value of *r* decreases as LCR increases, which is link idle probability plus the ratio of continuous packets without attacks. The final match probability p^k is the increase rate of *r*, and *p* is also calculated with TPR.

In practice, *r* reduces fast to *min* during the low network utilization periods and can stay around *min* for most of time; and when the matching occurs for a number of clock cycles, it returns to being a one-stage regular Bloom filter. The adaption speed depends on the control strategy, and then the average power consumption and the average latency for the adaptive Bloom filter can be obtained by replacing the *r* value in Equation 4-3 and 4-4.

- **Comparison of EABF to fixed stage schemes**

In simulation, two kinds of network conditions are considered: clean network data with TPR of 0 and a worse network status with TPR of 2.5%. The possibility of link idle or link working without attacks is high, we consider LCR = 90% in this section. PSR of [83] depends on the number of hash functions in the first stage. It can be a static value, e.g. 2, which is the same as *min* in EABF; or it can be different for different *k* values, e.g. $\lceil k/2 \rceil$, i.e. the hash functions are equally distributed in the first and second stages.

In comparison with the fixed two-stage scheme in [83], EABF uses Policy A for slow adaption. The number of hash functions *k* ranges from 0 to 35. Figure 4-19 shows the PSR comparisons of EABF and fixed two-stage scheme in [83]. As illustrated, PSR increases with *k*, the PSR of EABF is close to that of [83] for TPR=0

and EABF increase faster than [83] for TPR=2.5%. This is because EABF adapts its hash function distribution according to network conditions.

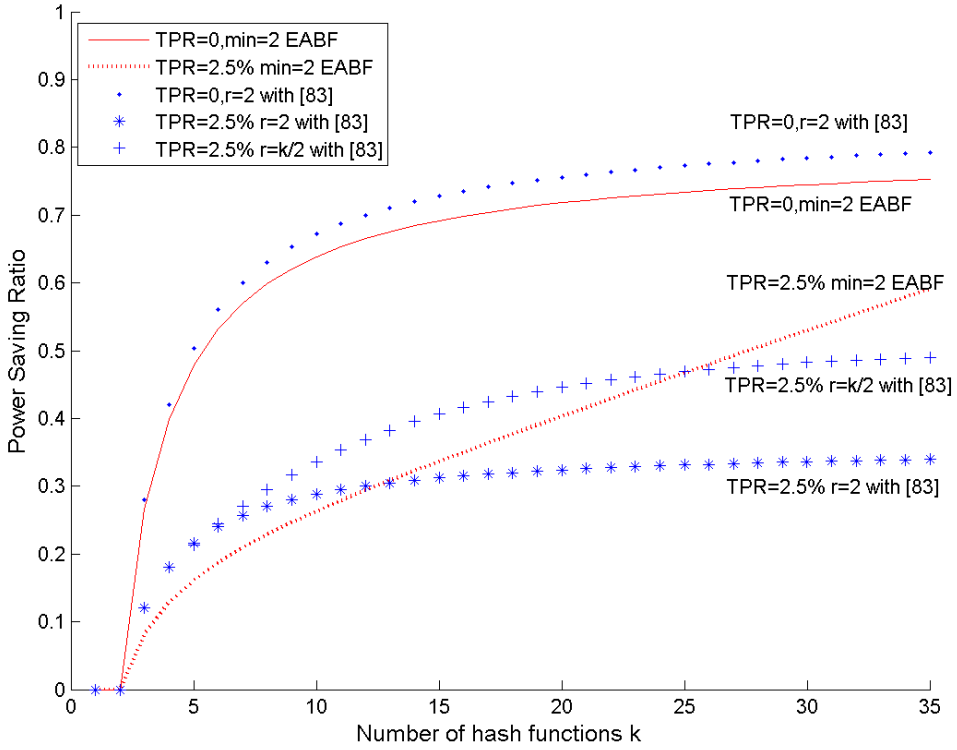


Figure 4-19 PSR Comparison of EABF and fixed two-stage Bloom filter

Figure 4-20 shows the PSR comparisons of two-stage EABF and the fully pipelined Bloom filter in [84] with one hash function per stage. Accordingly, we assume *min*=1, so the stable power efficiency state is the same as [84].

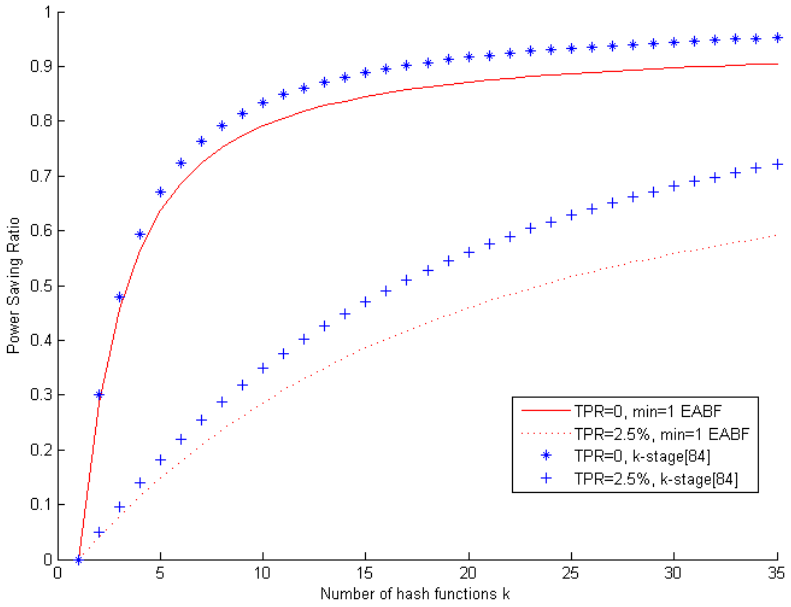


Figure 4-20 PSR comparisons of EABF and *k*-stage Bloom filter

Next the latency of fixed and adaptive Bloom filters is compared. As shown in Figure 4-21, for the above schemes for TPR=2.5%, the full-stage Bloom filter [84] has the longest latency of about 10τ, which is bad for high speed network; fixed two-stage scheme in [83] with static r=2 has latency of nearly 2τ when k gets larger, while the scheme in [83] with $r=k/2$ shows smaller latency; EABF shows the shortest average latency time, which is nearly 1τ since it can return to being a regular one stage Bloom filter when matches occur constantly.

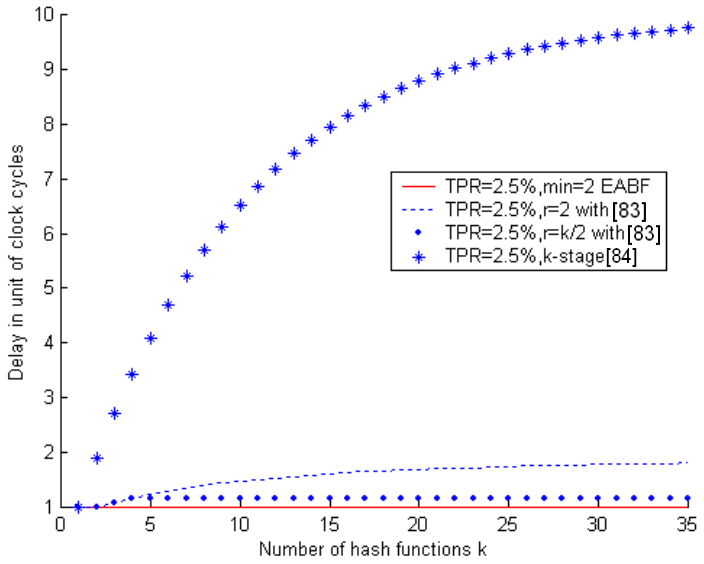


Figure 4-21 Latency of EABF and fixed stage schemes

In experiment, the adaptive EABF algorithm has a much smaller latency for malicious packets and a power saving ratio similar to that of fixed stage schemes for normal packets. Moreover, EABF with fast Policy B or blocked Policy C shows a smaller latency than that of slow Policy A, which will be shown in the experimental results.

- **Experiment Results**

EABF was implemented in Verilog HDL and simulated based on the Altera Stratix III FPGA for power estimation. Experiment parameters are based on those of the Policy C example. Power consumption was roughly estimated using Early Power Estimator (EPE) [71] based on resource utilization in the compilation report. The PowerPlay Power Analyzer [72] provides more accurate power data based on the simulated VCD file.

In the experiment, the hash table of EABF is trained with patterns from Snort rule set. In order to control the matching rate of Bloom filter, the input traffic is randomly generated with a controlled percentage of patterns to be matched (i.e. TPR). The packets that match any of the patterns are filtered, while the other packets pass.

Our Bloom filters use H_3 hash functions [29]. It is noted that different kinds of hash functions do not affect the power saving ratio so long as the value of (m/n) is the

same, where *m* is the length of the hash table and *n* is the number of the patterns. But the type of hash functions affects the power consumption due to the difference in memory usage. Since power consumption on FPGA also includes other modules, the total maximum power with *max* hash functions and minimum power with *min* hash functions are calculated in experiment, the power difference divided by (*max-min*) is the estimated power consumption of one hash function.

Table 4-3 shows the power consumption for Bloom filters with maximum and minimum number of hash functions for two frequencies, 125MHz and 62.5MHz. The estimated power consumption for each hash function is calculated by measuring the differences between the maximum and minimum power consumptions. From Table 4-3, it can be seen that the number of hash functions and the clock frequency affect the power consumption of the overall system, and the clock frequency affects the power consumption of each hash function.

Table 4-3 Hash function power estimation

Frequency (MHz)	Power (*mW*)		
	Max (*k* = 22)	Min (*k* = 4)	One Hash Function
125	870	523	19.3
62.5	727	412	17.5

A: Comparison with Fixed Stage Schemes:

In experiment, EABF runs the same testcases with the fixed stage Bloom filters [83][84] with TPR of 2.5%. During simulation of 10K clock cycles, the number of delays, and the number of active hash functions, which is related to the power saving ratio recorded to obtain the average delay and active hash function number associated with each *k* value. The latency results shown in Figure 4-22 basically follow Figure 4-21. Unlike fixed schemes, EABF can always maintain a small latency of nearly 1 clock cycle.

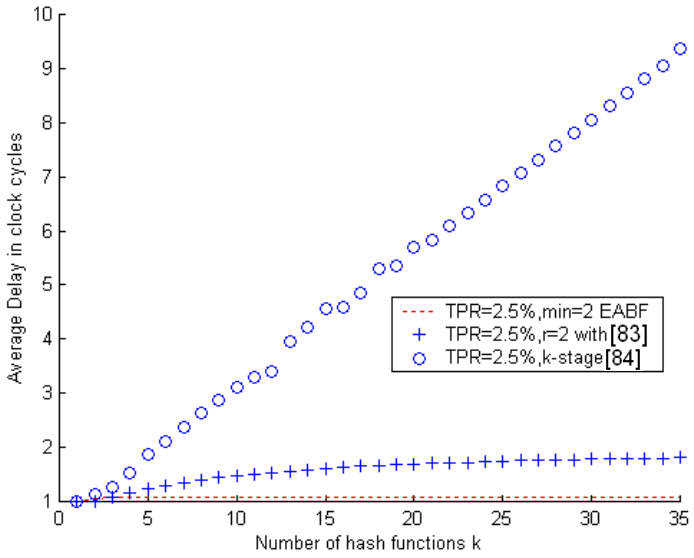


Figure 4-22 Latency comparison of EABF and fixed stage schemes

B: Comparison of control policies of EABF

The power consumption and latency behavior of EABF are determined by its control policy. The effects of three control policies are compared: Policy A for slow adaption, B for fast adaption with counters, and C for block adaption with *n* groups, each group uses a separate counter. Table 4-4 shows the minimum transition time between S_1 and S_2. The counter *c* is different for changing from S_1 to S_2 or from S_2 to S_1.

Table 4-4 Minimum adaption time of control policies

Minimum Adaption Time	A - Slow	B - Fast	C - Blocked
Without counter	*k-min*	1	*n*
With counter *c*	*(k-min)*c*	*c*	$c_1+c_2+..+c_n$

The average EABF latencies for TPR of 0 or TPR of 2.5% are all very small (i.e. only a little larger than 1), thus the comparison of the different policies is based on a larger TPR of 10%, as illustrated in Figure 4-23. As previous comparison with full *k*-stage Bloom filter, set *min*=1 for Stage I of EABF, which has longer delay due to larger FPR; and also set *min*=4 for comparison. The counter value determines the adaption speed. From Figure 4-23, larger counter value is more suitable for worse network conditions with larger TPR.

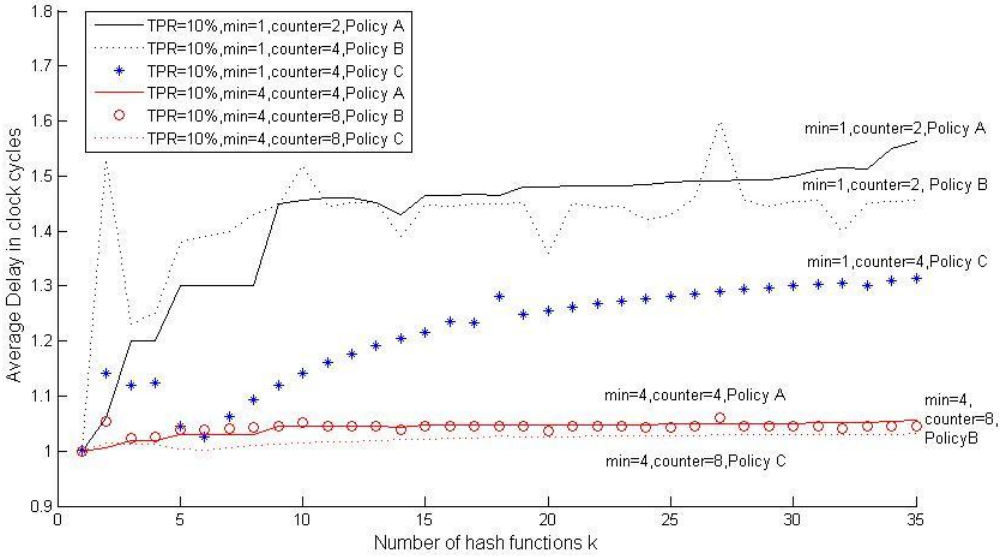


Figure 4-23 Latency comparisons of control policies

The delay time of Policy A increases with the number of hash functions *k* since more functions need to be moved before it reaches a stable state. Policy B is less related to *k* since it moves all the functions each time, although a counter is used for smoothing, it still fluctuates. By dividing hash function into several groups depending on their effects on FPR, Policy C presents a more stable control policy with a smaller delay time.

4.3 Multi-stage EABF

4.3.1 An overview of Method

The two-stage EABF can be extended to be a multi-stage adaptive Bloom filter, so that the number of effective stages and the number of hash functions in each stage can both be adjusted based on the current workload.

In this architecture, a Bloom filter with *k* hash functions includes maximum *N* stages, suppose each stage *i* could maintain at most $k_{i\text{-}max}$ hash functions, then the number of hash functions in each stage is k_1, k_2, …k_N, where $2 \le N \le k$, $k_1+k_2+\ldots k_N=k$, $1 \le k_1 \le k$ and $0 \le k_i \le k_{i\text{-}max}$ when $2 \le i \le N$. The two-stage platform is a special case of

multi-stage Bloom filter. When there are *k* stages, the multi-stage adaptive Bloom filter becomes the *k*-stage full pipelined Bloom filter in [84].

Current work in this section is based on software simulation. The hardware implementation of multi-stage Bloom filter will be done on FPGA platform in future work.

4.3.2 Adaptive Multi-stage Power Control Method

The two-stage platform of EABF could be extended to multi-stage. Different sleep and wakeup policies can be applied to hash functions in different stages. The following example is used to illustrate the multi-stage model.

The number of patterns is n=1K;
The number of entries in hash table is m=32K; then n/m=1/32;
The number of hash functions k that minimizes false positive rate f is

$$k_{opt} = (m/n)\ ln2 = 22.18 \approx 22;$$

False positive rate is $f \approx \left(1 - e^{-kn/m}\right)^k$, $f_{min} = 2.1035 * 10^{-7}$.

The division of stages is in line with the level of its false positive rate. In Figure 4-24, the log form y of false positive rate f, $y = -log(f)$, divides hash function number into four groups: y of Group 1 is smaller than 4, y of Group 2 is larger than 4 and smaller than 5, y of Group 3 is larger than 5 and smaller than 6, and y of Group 4 is larger than 6 and smaller than 7.

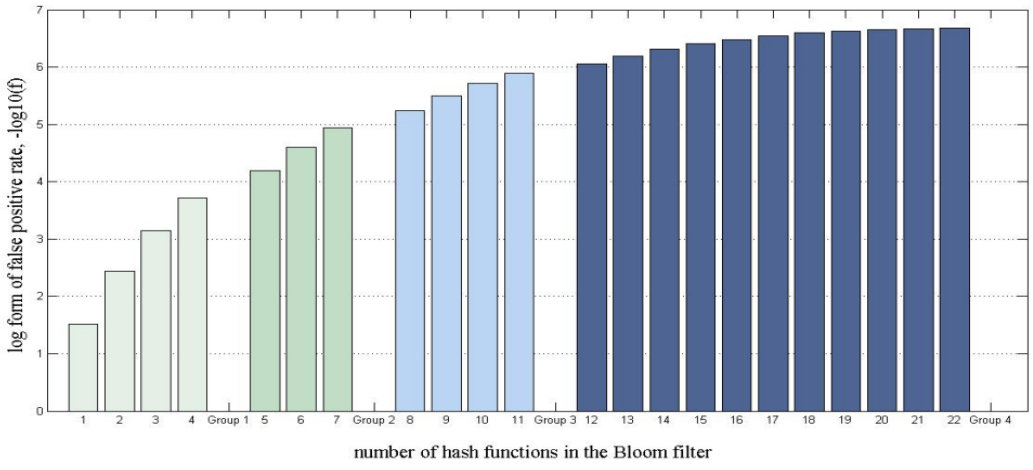


Figure 4-24 False positive rate with different number of hash functions

Figure 4-25 shows the control vector and hash function allocation for a four-stage model, where Stage II, III and IV associate with hash functions from different groups. The minimum number of hash functions is restricted based on their effect on the false positive rate. In the adaptive multi-stage Bloom filter, hash functions are linked to particular stages, and are moved between Stage I and the other associated stage. For example, HF_5 moves between Stage I and Stage II, HF_8 moves between Stage I and Stage III, and HF_{12} moves between Stage I and Stage IV.

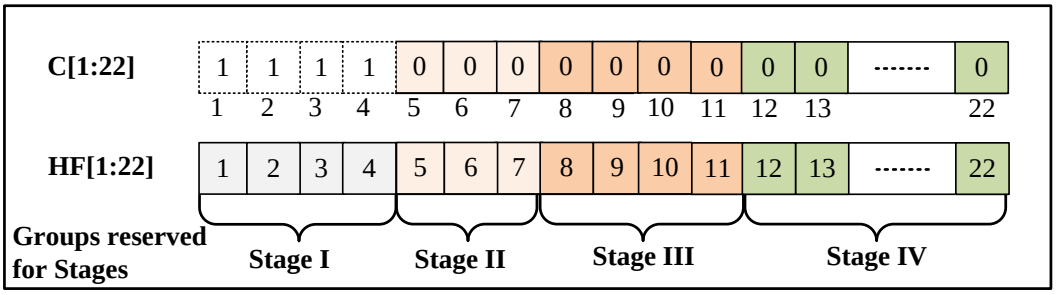


Figure 4-25 Multi-stage Bloom filter

Hash functions are moved out of Stage I in ascending order from 1 to *k-min* and moved back to Stage I in descending order from *k-min* to 1. Each hash function is associated with a specific stage when it is moved out of Stage I, e.g. in Figure 4-25, $HF_{4\text{-}7}$ belong to Stage II, $HF_{8\text{-}11}$ belong to Stage III and $HF_{12\text{-}22}$ belong to Stage IV.

In a multi-stage model, a latter stage is activated by a match output from its previous stage. Because the non-matched results are filtered and only a small percentage of normal traffic would incur a match, hash functions in latter stages have more chances to sleep. This model initially works as a regular Bloom filter, and gradually adapts to multiple stages. A stage is effective if at least one hash function is currently associated with this stage.

Functions belonging to Stage I always play an active role. Functions belonging to other stages can sleep when network packets are filtered by their previous stages, or work when there are incoming packets to their stage buffers. A latter stage offers its hash functions a larger opportunity to sleep.

Figure 4-26 shows the possible states for a hash function in each clock cycle:

active in Stage I, active in Stage II, III, IV or sleep. Each hash function is controlled by a control bit C[*i*], value 1 means HF_i is in Stage I and value 0 means HF_i is in Stage II, III, IV, depending on which stage it is related to. When C[*i*] = 0, HF_i sleeps except when Stage I produces a positive match. When C[*i*] = 1, HF_i is brought back to Stage I.

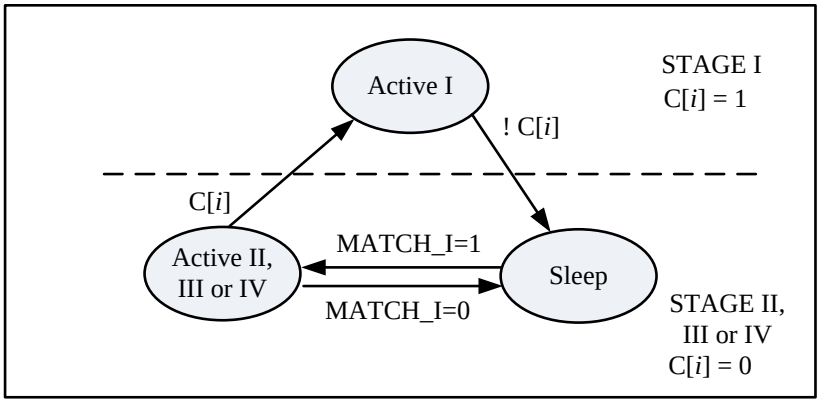


Figure 4-26 Hash function state in multi-stage Bloom filter

Essentially, a hash function adapts its state by observing Bloom filter input text and the match results. An invalid input means there is no effective traffic load. This corresponds to low traffic conditions or large packet intervals. Moreover, when there is continuous data input without a match for a long time, it probably belongs to normal traffic. Since the Bloom filter has no false negative, less hash functions can handle the packet inspections. Then some hash functions are gradually moved to sleep state in Stage II, III, and IV.

Similar to two-stage EABF, the ***easy move-in and hard move-out*** principle is used in order to prioritize performance. Hash functions in latter stages are more easily moved back to Stage I and are harder to move out to the latter stage. For instance, if there is a matched string, one hash function in Group IV is moved back to Stage I, whereas if there are 4 continuous matched strings, one hash function in Stage II is moved back to Stage I. In this case, it takes one clock cycle to move in to Stage I and four clocks cycle to move out from Stage I.

A match output means a pattern is recognized. Since malicious attacks like

viruses appear frequently during certain periods in network applications, hash functions need to be brought back to Stage I quickly in order to check the subsequent packets.

Another scheme is that a hash function can be moved from Stage I to Stage II, and continuously to Stage III or Stage IV. As an example of this case, the control vector C[*i*] may use two bits to adapt a hash function among four stages, where an increment of the value takes the hash function to a later stage, and a decrement of the value brings the hash function to an earlier stage.

Compared with the fixed two-stage scheme in [83] or the fully pipelined scheme in [84], the multi-stage adaptive Bloom filter exhibits different structures corresponding to various traffic conditions. Depending on certain control policies, it adapts to workloads for a balance of larger power saving ratios and shorter average latency. The adaptive scheme holds the structure as a fixed stage scheme, when the network condition is stable. Compared to the two-stage EABF, the multi-stage adaptive scheme can be more flexibly configured with even larger power savings.

4.3.3 Control Strategy for Self-adaption

In principle, hash functions in Stage II or latter stages have more chances to sleep, leading to larger power savings and longer delays. The movements of hash functions are determined by the control strategy. Here we define the "*moving-in*" as moving a hash function to an earlier stage, i.e. moving from other stages to Stage I or moving from Stage III to Stage II; and the "*moving-out*" as moving a hash function to a later stage, i.e. moving from Stage I to stages II, III, IV.

Basically, the "*moving-in*" action is triggered by a match result, and the "*moving-out*" action depends on two criteria: 1) the detection of invalid inputs for Bloom filter; 2) non-match for a period of time. Firstly, invalid inputs appear when the Bloom filter is currently idle with no work load. Secondly, when there is continuous data input without a match for a long time, it probably belongs to normal

traffic. A time counter for the number of filtered packets is used to record the number of clock cycles for clean traffic.

Frequent adaption of hash functions has a delay overhead for the Bloom filter. For smoother adaption, the "*moving-in*" and "*moving-out*" actions also refer to a specific counter value. Since performance is more important than power for network applications, the "*moving-out*" counter is larger than the "moving-in" counter following *easy move-in and hard move-out* principle for performance priority. Moreover, when a hash function is moved out of Stage I for low power, it can be moved to several possible stages, e.g. stages II, III or IV. But when it is required to be moved back for fast response, it is moved directly to Stage I. This is because the malicious attacks like viruses appear frequently during certain periods in network application, and so hash functions need to be brought back to Stage I quickly in order to check the subsequent packets.

Similar to the two-stage EABF, the multi-stage adaptive Bloom filter also has two special states: performance optimization state S_1 defined as simple Bloom filter that the hash functions has all in Stage I, and power optimization state S_2 defined that each stage reserves the minimum number of hash functions except the last stage.

Four control strategies are proposed for multi-stage adaption: 1) cascading stage adaption; 2) assigned stage adaption; 3) fast speed group adaption; and 4) variable speed group adaption.

1) Policy A: Cascading Stage Adaption

When a hash function is moved out of Stage I, it goes to Stage II, and then Stage III and Stage IV, stage by stage. In particular, a hash function is moved to Stage I when there is a matched result; and a hash function is moved to the next stage, e.g. from Stage I to Stage II or from Stage II to Stage III, when the system is idle, as observed from invalid inputs, or the system has been clean for some time, as seen from the non-match results.

This policy slowly controls the number of stages in the system. Considering the

traffic condition that causes a continuous one-direction movement of hash functions, based on the example in Figure 4-25 using Policy A, EABF structure requires 43 "*moving-out*" shifts to move from performance optimization state S_1 to power optimization state S_2. On the other side, EABF structure requires 18 "*moving-in*" shifts to move back from power optimization state S_2 to performance optimization state S_1. With this policy, two control bits are needed to represent the four possible stages that a hash function is linked to. At the same time, the control circuit for a hash function becomes more complex, which brings a larger overhead of power dissipation.

2) Policy B: Assigned Stage Adaption

As shown in Figure 4-25, considering the number of hash functions is fixed, when a hash function is moved out of Stage I, it will go to a particular stage whose previous stages are fully occupied. Given that a hash function has two possible working stages, one control bit can handle it. This policy provides fast adaption to multi-stages and uses simpler control circuit.

This policy moves to later stages faster with a simpler control circuit. The adaption of controller *C* is the same as that in Figure 4-13 for two-stage EABF, and a hash function works in Stage I when its control bit *C*[*i*] equals to 1, or it works in Stage II, III or IV when its control bit *C*[*i*] equals to 0. The "*moving-out*" preferentially moves functions with larger suffixes to the first non-full stage. For "*moving-in*", functions with smaller suffixes are moved preferentially from the last working stage.

3) Policy C: Fast Speed Group Adaption

Based on the group division in Figure 4-24, one movement of hash functions can be performed on a group, so that more than one hash functions in the same group is moved each time. This method can be used with Policy A or Policy B. For example, under Policy A, when a matched result occurs, the 11 hash functions of Group 4 are shifted back from their current stages, and then the next matched result brings 4 hash

functions of Group 3 back to Stage I. On the other hand, for idle network conditions, it shifts all Group 2 hash functions to Stage II.

4) Policy D: Variable Speed Group Adaption

Functions in different groups affect the false positive rate in different scales. Instead of a fixed speed, variable speed can be used for groups. In line with Figure 4-25, for example, four continuous invalid inputs trigger a "*moving-out*" shift of Group 2. Similarly, 8 continuous invalid inputs trigger a "*moving-out*" shift of Group 3 and 16 continuous invalid inputs trigger a "*moving-out*" shift of Group 4.

4.3.4 Optimization with Feedback to Dynamic Frequency Control

The power efficiency of the two-stage and multi-stage EABF can be further optimized with the implementation of dynamic frequencies. As shown in Figure 4-27, the Bloom filter engine can run at different frequencies and the clock selection signal (*clkselection*) is determined by the feedback of the engine status. When the Bloom filter stays in power optimization state S_2 for a period, it feeds back to clock control block for a lower frequency; when the Bloom filter stays in performance optimization state S_1 for a period, it feeds back for a higher frequency.

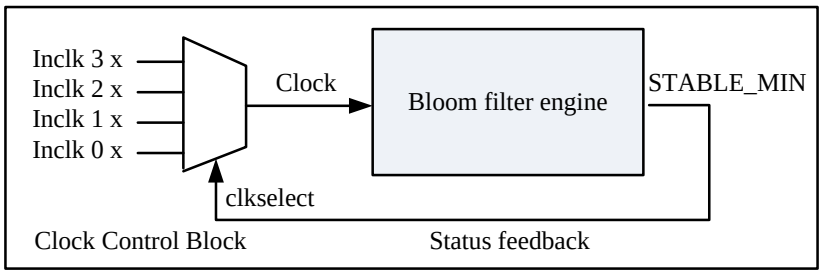


Figure 4-27 Feedback to frequency control

Usually, multiple Bloom filters are used in parallel. Figure 4-28 shows a pattern matching system with multiple parallel Bloom filters, where L_{min} and L_{max} are the minimum and maximum length of text to be processed by a Bloom filter. Depending on workload status and match feedbacks, the Bloom filters are controlled with the

same signal. The clock control unit starts when the control vector reaches a stable status and it changes the running frequency of the Bloom engine if necessary. The power consumption is the sum of all Bloom filters.

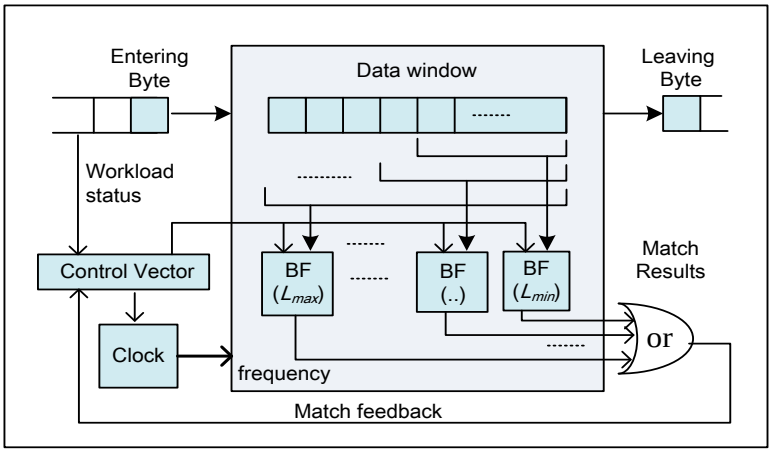


Figure 4-28 Feedback to frequency control

4.3.5 Analysis and System Evaluation

- **Delay and Power Savings for Multi-stage Platform**

The platform starts from performance optimization state S_1, as a regular Bloom filter, and it gradually adapts to power optimization state S_2. Consider an *N*-stage Bloom filter which has *k* hash functions distributed as k_1 in the first stage, k_2 in the second stage and k_N in the last stage, where $N<k$ and $k_1+k_2+\ldots k_N=k$.

Suppose the power consumption of a hash function is P_H and the average matching probability of a hash function is *p*, whose optimal value is 0.5 for random input. The power consumption of an *N*-stage Bloom filter is shown in Equation 4-5, where k_i is a dynamic value in an adaptive system.

$$P_{\text{N-stage}} = k_1 \cdot P_H + p^{k_1} k_2 \cdot P_H + p^{k_1+k_2} k_3 \cdot P_H + \ldots + p^{k-k_N} k_N \cdot P_H \quad (4\text{-}5)$$

Compared to a regular Bloom filter, PSR of multi-stage Bloom filter at a given state is shown in Equation 4-6, where the definition of PSR is the same as that of a two-stage EABF in Equation 4-3, calculated as 1 minus the ratio of the average number of active hash functions ($P_{N\text{-}stage}$) and the total number of hash functions (*P*).

The k_1 hash functions in the first stage are always active; the k_2 hash functions in the second stage become active if the input is matched by the k_1 hash functions in the first stage, and so on for k_N in N-stage.

$$\begin{aligned} PSR &= \left(1 - P_{N-stage}/P\right) \cdot 100\% \\ &= \left(- 1k_1 + (p^{k_1}k_2 + p^{k_1+k_2}k_3 + \quad + p^{k-k_N}k_N \quad k\right) \cdot) / \end{aligned} \tag{4-6}$$

For delay estimation, it is assumed that each stage has the same latency τ. The match latency is the maximum latency of an N-stage Bloom filter,

$$T_{N\text{-}max} = \tau + (N\text{-}1) \cdot \tau = N \cdot \tau \tag{4-7}$$

The average initial latency for random input is shown as follows. Besides, as the number of matching increases, the multi-stage EABF gradually returns to one-stage Bloom filter and the latency decreases to τ.

$$T_{N\text{-}avg} = \tau + (p^{k_1} + p^{k_1+k_2} + \ldots + p^{k-k_N}) \cdot \tau \tag{4-8}$$

A special case of the multi-stage Bloom filter is the fully pipelined k-stage Bloom filter, which has one hash function per stage. In this case, the PSR and average latency are simplified as follows.

$$PSR = \left(1 - \frac{1}{k} \cdot \sum_{i=1}^{k} p^{i-1}\right) \cdot 100\% \tag{4-9}$$

$$T_{latency} = \tau + \tau \cdot p + \ldots + \tau \cdot p^{k-1} = \tau \cdot \sum_{i=1}^{k} p^{i-1} \tag{4-10}$$

● Comparison of multi-stage EABF to fixed stage schemes

In simulation, the multi-stage EABF is compared with the two-stage EABF and fixed stage schemes in [83] and [84]. The simulation of multi-stage EABF is similar to that of two-stage EABF, which inspects the input traffic with the pre-built hash tables. The difference is that more stages are deployed for better flexibility. The parameters are based on the example in Section 4.3.2. The 22 hash functions are allocated to a four-stage EABF according to Figure 4-25. The adaptive EABF uses Policy A of slow adaption. The patterns from Snort rule set are used to train the hash table and generate the input traffic. The traffic trace is randomly generated with a TPR of 2.5%, that is, 2.5% of traffic will lead to the positive match of the Bloom filter. The

match results and the number of clock cycles used for each matching result are processed by a statistical program.

Table 4-5 shows the PSR and latency comparisons. The latency is in unit of clock cycles. For TPR=2.5%, the fully serial *k*-stage Bloom filter [84] has the best PSR of 65.0% at the cost of the longest latency of 9.10 clock cycles. The 2-stage and 4-stage EABF obtains larger PSRs (e.g. 53.5% and 59.2%) and smaller latencies (e.g. 1.08 and 1.24 clock cycles) than that of the fixed 2-stage Bloom filter [83] (e.g. 46.5% PSR and 1.54 clock cycles). The reason for this improvement is that EABF adapts its hash function distribution with traffic conditions. Comparing the 2-stage and the 4-stage EABF, the 4-stage EABF is more flexible, which has a better PSR with a small increase of latency.

Table 4-5 Comparisons of EABF and fixed stage Bloom filters

Comparison (TPR=5%)	Fixed Bloom filter		EABF (*min*=4)	
	2-stage	*k*-stage	2-stage	4-stage
Settings on each stage	(11, 11)	(1,1..1)	(4, 18)	(4, 3, 4, 11)
PSR	46.5%	65.0%	53.5%	59.2%
Latency	1.54	9.10	1.08	1.24

4.4 Cache Acceleration of CBF for Precise Matching

4.4.1 Motivation and Related Work

While most Bloom filter applications tolerate imprecise matching for searching a large data set, there are situations that require the exact matching. For precise matching in routing protocols, H. Song [86] uses an extended Bloom filter to verify if the suspicious matching packets really contain a pattern. The false positive analysis is achieved by extending the Bloom filter with an additional link of signatures. However, it does not mention how to handle the increased processing latency generated with this scheme.

A separate table can be added in the Bloom filter for precise matching, which keeps all the mapped patterns for each entry with a non-zero counter. When a string

results in a match by mapping to all of non-zero entries in the hash table, it needs to be compared with the entry's pattern list. It is a "*false match*" if the string does not exist in the pattern list. In fact, the precise matching only needs to compare with the pattern list of the entry with the smallest counter, rather than compare with the pattern list of all the mapped entries.

However, in high-speed network applications, the huge pattern set usually mean that FPGA memory is insufficient. Thus the pattern table needs to be allocated off-chip. The drawback of this solution is the extra time and energy of accessing off-chip memory. Borrowing the idea of multi-level storage in computer memory organization, a block of registers or RAM can be used as a cache on the FPGA. The processing engine searches the cache first, and a cache-line with a number of patterns is brought in for each cache miss. For better cache replacement, continuous table entries have to be as relative and compressed as possible. Moreover, the inherent characteristics in patterns is explored for a better organization of the pattern table, so as to achieve a higher cache hit rate.

In order to be memory, time and energy efficient, the contributions of this section lie in three aspects: firstly, the compressed CBF structure and its update algorithm; secondly, pattern grouping based on the relativity of patterns in network packets; thirdly, on-chip cache structure designed with cache index and cache block, and proper replacement policy for higher cache hit rate.

4.4.2 System Overview of CBF with Cache

A regular Bloom filter with one-bit wide array provides fast lookup of traffic. CBF with counters is utilized to support the insertion or deletion of patterns, but it is not as fast as Bloom filter. The pattern matching process involves frequent hash table lookup operations. As a balance, the traditional Bloom filter is used for lookup operations and CBF is used for update operations. Each non-zero bit in the Bloom filter corresponds to a counter in CBF while all zero bits in the Bloom filter are also zero in CBF. In the case of a hardware platform for high-speed network processing,

memory is a limited resource. Therefore, an on-chip Bloom filter table for pattern matching is used together with an off-chip CBF table for pattern verification. For precise matching, each counter is associated with a pattern link similar to the link list in [86]. The update of the counter array in CBF modifies its associated pattern array at the same time.

But the memory saving will seriously reduce the processing speed, because the off-chip access latency is high, which is about 10 FPGA clock cycles. As a compromise, a special on-chip area is used in this work as a cache and propose Cached CBF (C^2BF). Initially, the verification needs to access the off-chip memory directly. Assisted by the cache, the verification first searches in the cache and accesses the off-chip memory if the cache returns a cache-miss. Figure 4-29 shows an overview of C^2BF and its matching verification process. The Bloom filter and cache are kept in on-chip memory, while CBF tables are located off-chip. T_1 is the counter table, whose non-zero counters are connected with the pattern table T_3 stored in off-chip memory. The pattern sequences stored in pattern lists point to the full patterns in T_3.

Suppose n=5K patterns are mapped to a hash array of m=80K entries by k=8 hash functions, as shown in Figure 4-29. Since the pattern size of 10 bytes is large and k copies of the patterns are redundant, the associated items such as P_{21} are their offset values in T_3, and they are 13 bits wide for n=5K. It can be seen that the off-chip CBF array T_1 and Pattern Table memory T_3 are much larger than an on-chip regular Bloom filter array.

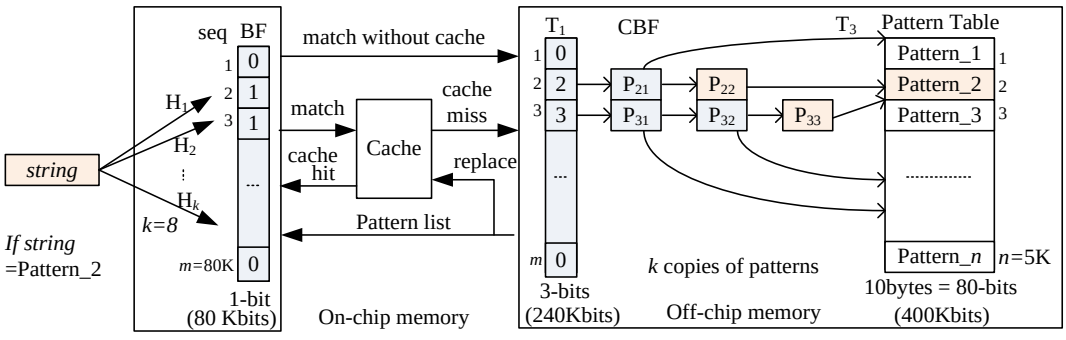


Figure 4-29 System overview and the matching verification

The matching verification process queries the pattern list associated with the hashed entries. Using cache, it first searches the contents of the cache and returns the pattern list for a cache hit; or else for a cache miss, the pattern list is acquired from off-chip memory and a cache line is refreshed. The input string is Pattern_2 in Figure 4-29 and is match by the Bloom filter. For verification after a cache miss, it searches T1 and T3 and reports a valid match.

The matched pattern can be connected by a pointer from the corresponding link list. But link list is not straightforward for implementation in FPGA designs. Since CBF is primarily built off-line, a compressed pattern array is used instead. After the building of the CBF table and link list are built, we can put the associated patterns one by one in an associative pattern list table T_2. Figure 4-30 shows an example, T_1 stores a start address next to each counter, address 0 for entries with counter 0, 1 for the first item in T_2 and so on. T_2 stores a tag and pattern offset in T_3. The tag is "1" for the indication of last item with the current counter or is "0" otherwise. Since k copies for each pattern are mapped to CBF, there are $k \times n$ = 40K entries in T_2, where k is the number of hash functions and n is the number of patterns.

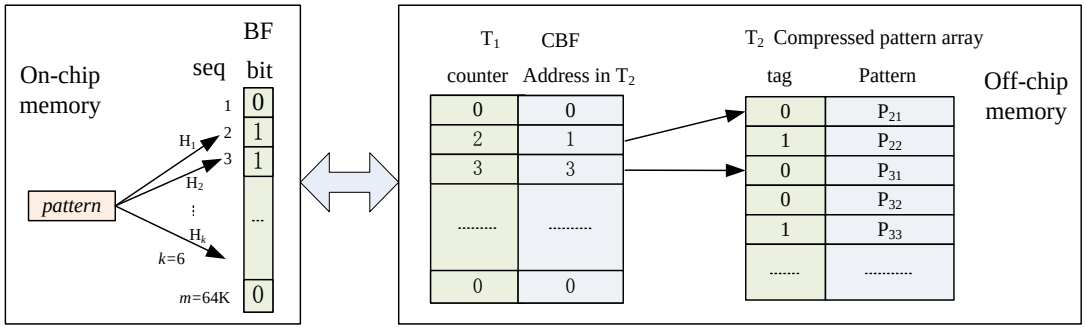


Figure 4-30 CBF and pattern array structure in off-chip memory

4.4.3 Update of CBF and Pattern Array

After initial set up, the update operation will affect the T_1, T_2 and T_3 tables in off-chip memory. Firstly, if a pattern is to be deleted, the CBF deletion operation finds the hashed entries, decrements their counters by one in T_1 and deletes the corresponding pattern in T_2. If the counter is greater than 1, the "1" tag for the end of

this entry should also be modified if necessary. For network security applications, the deletion operation normally means deleting a recognized characteristic string out of the dataset, which rarely happens. Based on Figure 4-30, "Pattern_2" is deleted in Figure 4-31. Considering the number of deleted items should not be very large, we do not further adjust the T_2 or T_3 table by filling the space freed by the removal of *k* or 1 copies of the deleted pattern. Instead, a deletion table T_4 is maintained, which keeps the invalid entry addresses in T_2 and pattern table T_3. T_4 is a sorted table in ascending order and is referenced for insertion operation. In Figure 4-31, tags of P_{21} and P_{22} are set to 1, the positions of empty spaces P_{22} and P_{33} are recorded in T4 as (2, 5).

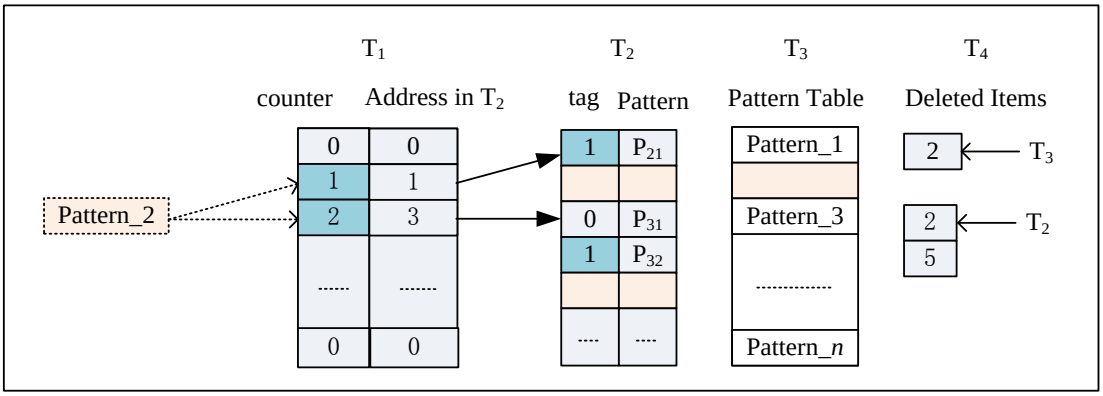


Figure 4-31 Table structures after deletion of Pattern_2

Secondly, CBF insertion operation is triggered when a new pattern "Pattern_*new*" is added. If T_4 is not empty, it chooses the last item which is also the smallest pattern number N_{min}, and adds Pattern_*new* in entry N_{min} of T_3. Otherwise, Pattern_*new* is added at the end of T_3. Pattern_*new* is mapped to *k* entries and their counters are incremented by one in the CBF table. Based on Figure 4-30, Figure 4-32 shows an example of inserting Pattern_*new* in entry 2. The tag of P_{22} is modified to "0" and the original P_{22} points to the new position of P_{22}, the next of which is the inserted new pattern Pattern_*new*. However, this kind of operation takes more time since the lookup of pattern P_{23} needs to go through a link. In network security, new patterns are added from time to time, but too many links lead to a confused structure.

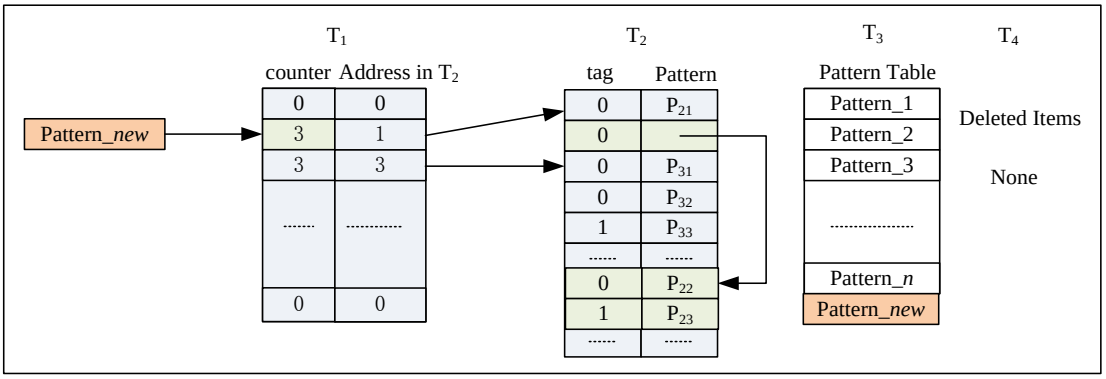


Figure 4-32 Insertion operation of Pattern_*new* on entry 2

Another method is to divide the whole table into multiple sub-tables, and reserve a number of empty items for new patterns between two sub-tables. Figure 4-33 shows a sub-table for entries 1 to 50. When a new pattern P_{23} is to be inserted into entry 2, all the items between the start address "3" of entry 3 and the reserved position are moved backward one entry in parallel to leave room for P_{23}. The moving operation can be performed in parallel in one clock cycle. Moreover, all the T_2 addresses for non-zero entries within 1-50 in T_1 need to be incremented by one. This kind of table retains a compact and clear structure.

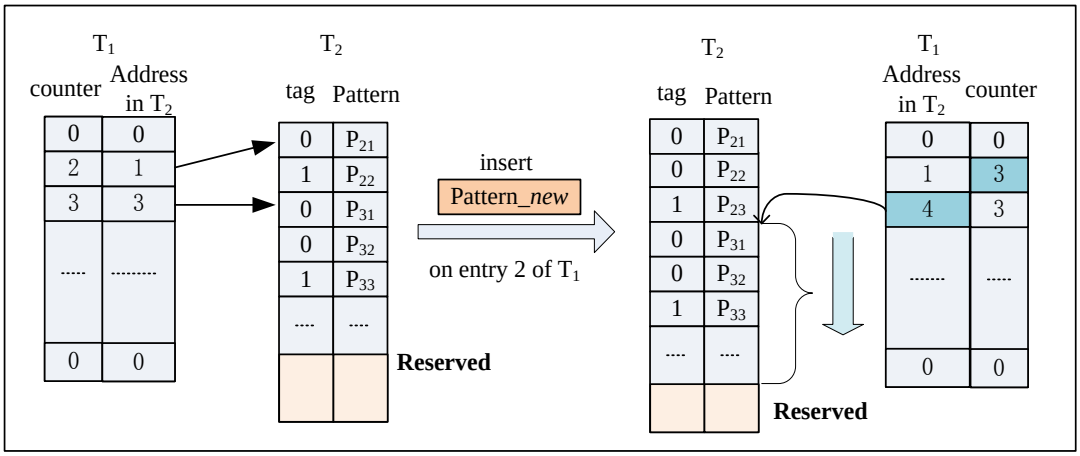


Figure 4-33 Insertion operation with reserved space

4.4.4 Pattern Grouping and Cache Design

The cache mechanism takes advantage of pattern locality detected in matched network packets. Some patterns have similar functionality to describe the characteristics of certain events. Therefore, instead of using a random pattern array in

T_3, we preliminarily divide patterns into groups. Take the Snort rule set [9] as an example, it consists of categories including *ftp*, *dns*, *dos*, *P2P*, *Trojan Horse* etc.

Consecutive sequence numbers are assigned to patterns that belong to one group. The group size depends on the unit of cache replacement, i.e a cache line. In addition to the concern of pattern category before building a pattern table, we can also train the table with the matching of real network traffic. The matched packets mostly belong to various kinds of malicious traffic. During a short-term period, the malicious network flows are possibly related and they might attack repeatedly, such as Dos or ARP flooding. This characteristic brings opportunities for cache. During CBF lookup for a time period, a history table is used to record all the matched patterns one by one. This table can be analyzed off-line to discover potential relativities for better pattern grouping.

The verification process without any cache mechanism works as follows. If there are *k* hash functions, when a text produces a match in the Bloom filter, it maps to *k* entries in T_1, and the one with the minimum counter value is chosen. Then it looks up the T_2 table to locate the pattern numbers associated with this address. Next it looks up the T_3 table with the pattern number and returns each of the patterns to be compared. This process stops when the returned pattern is the same as the text, which means a true positive; or it stops when all of the patterns have been brought in, which means a false positive. The maximum possible delay time is $(2 \times max\_counter \times \tau_{off\text{-}chip})$. However, $\tau_{off\text{-}chip}$ is about 10 FPGA clock cycles. The objective of the work in this section is to decrease the verification delay using proper cache design.

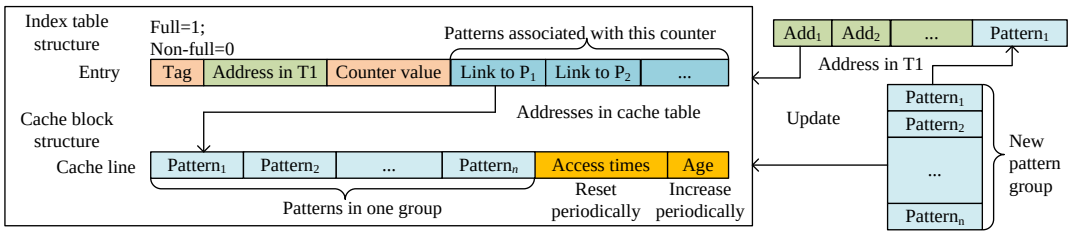


Figure 4-34 Cache organization

The left part of Figure 4-34 shows the cache index and cache block structure.

The index table indicates if at least one of the associated patterns for an entry in T_1 has been brought to cache. An index entry includes a full tag, counter address in T_1, counter value and links to addresses in the cache block for its associated patterns. By comparing the counter value with the number of attached pattern links, the tag indicates whether all of this counter's associated patterns are in cache, 1 for full and 0 for non-full. For a simpler search, the cache index is organized in such a way that the T_1 addresses are in ascending order. The cache-line is the basic replacement unit in the cache block. It is known that the T_1 and T_2 tables are randomly distributed due to hash characteristics, while the T_3 table can be carefully organized using pattern groups. Accordingly, a cache-line is comprised of a pattern group, access times and age. The latter two items are designed for cache replacement. The access times field is the number of times that the cache-line is accessed. And the age field is incremented periodically to indicate how long the cache-line has been in cache.

When a string is preliminarily matched, the mapped addresses in the simple Bloom Filter hash array are also the addresses in T_1. We first check whether one of these addresses exists in the cache index. If none of them appears, it produces a cache miss and searches off-chip memory. If one of its addresses has full tag, all its linked patterns in the cache block are searched and compared in parallel. At the same time, the access times of their groups are increased by one. If none of the addresses in the cache index is associated with a full tag, it also compares all of the linked patterns; however, if none of the patterns are equal to the input string, it still produces a cache miss and searches the off-chip memory.

When a cache miss occurs, and so a pattern is brought on-chip for comparison with the input string, then its pattern group is also brought to cache to write a new cache-line. As shown in Figure 4-34, the pattern to be brought into cache associates with k mapped counter addresses in T_1. The update process includes cache index update and cache block update. For a cache index update, if Add_1 of $Pattern_1$ is already in the index, the link to $Pattern_1$ address in the cache block is attached to Add_1 entry, whose tag should also be updated. Otherwise, a new entry is inserted in cache

index. For cache block update, the pattern group is brought to a new cache-line if there is an available cache-line.

If the cache is full, we need to replace a cache-line with the new pattern group. Considering similarity in short term network traffic, we use the Least Recently Used (LRU) principle as the cache replacement policy. In particular, it can choose cache-lines with the smallest access times. As shown in Figure 4-34, the accessed times field needs to be reset periodically since network locality only stands for short term traffic. Otherwise, the early frequently accessed cache-line would never be replaced. If there is more than one cache-line, choose the one with the largest age. Moreover, the LRU policy can also be implemented in another way with age priority. It first chooses cache-lines with the largest age, and then chooses the one with the smallest access times.

4.4.5 Analysis and Experiment Evaluation

We first analyze the average length of a pattern list associated with each counter. A Bloom filter uses k independent hash functions to map each input string to an array of m bits, which is initially trained with n patterns. The false positive rate f is given as $f \approx \left(1-e^{-kn/m}\right)^k$ and the parameter k that minimizes f is $k_{opt} = (m/n)\ ln2$ [12].The probability that a counter equals to i in its corresponding CBF is shown in Equation 4-11.

$$\begin{aligned} f_{counter=i} &= \binom{n}{i} f_{hit}^{i} f_{miss}^{n-i} \\ &= \frac{n!}{i!(n-i)!}\left[1-\left(1-\frac{1}{m}\right)^{k}\right]^{i}\left(1-\frac{1}{m}\right)^{k(n-i)} \\ &\approx \frac{n!}{i!(n-i)!}(1-e^{-ki/m})e^{-k(n-i)/m} \\ &= \frac{n!(e^{-k(n-i)/m}-e^{-kn/m})}{i!(n-i)!} \end{aligned} \tag{4-11}$$

Song etc. in [86] shows that theoretically, the counters of value 1 are more than 99% among all non-zero counters. Then for each matching verification, it requires

only one access to CBF arrays T_1, T_2 or T_3.

Targeted at hardware implementation on FPGA, the C^{2}BF system was written in Verilog HDL and simulated in Modelsim. For compact off-chip memory design, the three hash functions illustrated in [87] are used for comparison, including H_3, BIT and XOR. Then a hash function that generates a smaller counter value can be chosen, for both the average and maximum values. Similar to the example in Figure 4-29, the number of hash functions k =8, the number of patterns n is 5K, and the number of entries in hash table m is m_1=80K or m_2=100K, the false positive rate is $f_1 = 5.7\times10^{-4}$ or $f_2 = 1.4\times10^{-4}$. The minimum required off-chip memory and counter distribution of three hash functions is shown in Table 4-6. The average value for non-zero counters is slightly more than 1, and the match verification requires on average about one access to off-chip memory. Comparatively, the H_3 hash function has better distribution for counters in T_1.

Table 4-6 Counter distribution in CBF table and required off-chip memory

m	Off-chip memory	Non-zero counters by H3			Non-zero counters by BIT			Non-zero counters by XOR		
		Ratio	Avg	Max	Ratio	Avg	Max	Ratio	Avg	Max
m_1=80K	2.48Mbits	48.3%	1.008	5	46.9%	1.03	7	47.2%	1.015	6
m_2=100K	2.86Mbits	39.0%	1.003	4	37.8%	1.019	4	38.2%	1.005	5

To evaluate the cache design, the total processing time of the Bloom filter for precise matching, with or without the cache mechanism, is compared. We use 40Kb on FPGA as the cache block for the size of 500 maximum patterns ($500\times10\times8$ = 40Kb). The size of a cache-line is related to that of a pattern group. During cache replacement, patterns of one group are brought to a cache-line. Cache hit rate can be increased since the same kind of traffic flows appear closely. Suppose that the average cache hit rate is around 50%, the off-chip access time is 10 FPGA clock periods, and the cache block contains 50 cache-lines, each of which has 10 patterns. In order to test the system under different traffic conditions, network packets with different true positive rates are simulated, which are the pattern matching probabilities, including 0.1%, 1% and 10% and 50%. Table 4-7 shows the processing time comparison for precise matching with or without cache design.

Table 4-7 Processing time comparison with or without cache mechanism

Matching probability	*0.1%*	*1%*	*10%*	*50%*
Average processing time without cache	1.010	1.100	2.008	6.010
Average processing time with cache	1.007	1.065	1.650	4.003
Reduction ratio	0.35%	3.18%	17.5%	33.5%

Based on cache size, larger cache-line size corresponds to lower cache-line numbers, which means that more patterns can be brought in each time, but there are less pattern groups in cache. Three schemes are compared with the same total cache size: 25 cache-lines with 20 patterns as in Scheme 1, 50 cache-lines with 10 patterns as in Scheme 2, 100 cache-lines with 5 patterns as in Scheme 3. The cache-hit ratio also depends on the traffic relativity of continously matched packets. If a pattern of Group *n* is matched, the probability that one of the next few matched patterns also belongs to Group *n* is defined as traffic relativity. The cache hit ratio using two LRU policies is compared, with access times priority or with age priority, for cache replacement.

Table 4-8 Cache hit rate with different schemes under different traffic conditions

Schemes	*Scheme 1*				*Scheme 2*				*Scheme 3*			
Traffic relativity	40%		80%		40%		80%		40%		80%	
LRU priority	age	acc	age	acc	age	acc	age	acc	age	acc	age	acc
Cache hit rate	48%	45%	75%	73%	55%	51%	91%	86%	53%	53%	83%	81%
Redution ratio*	38%	35%	65%	63%	45%	41%	81%	76%	43%	43%	73%	71%

*it is the reduction for matching verification time compared to traditional scheme without cache.

Table 4-8 compares the cache hit rates and reduction ratios of verification time. The cache hit rate can reach more than 80%; compared to a traditional scheme without cache, it reduces more than 70% of the verification processing time with cache design. Moreover, LRU replacement policy with access times priority has a higher cache hit rate and is more suitable for network applications.

4.5 Summary

The Bloom filter is a memory efficient method for the implementation of DPI. This chapter describes the use of Bloom filters for power efficient hardware

acceleration of DPI. On the other hand, regular Bloom filters do not support the deletion of patterns, which can be solved by CBF with a larger memory size for counters in hash table.

This chapter presents four works on Bloom filters. Firstly, a multi-pattern matching system using parallel Bloom engines for high speed pattern matching is proposed, and an optimized CBF structure is used for memory reduction. A pattern matching algorithm is implemented in parallel based on CBF.

Secondly, a adaptive Bloom filter model EABF is designed for a balance of power efficiency and performance for network applications. Depending on a control policy, EABF moves hash functions among three states: working in Stage I, working in Stage II or sleep in Stage II. According to the current workload, EABF maintains the minimum number of hash functions in Stage I for idle or normal network conditions. It can also revert quickly to the regular Bloom filter structure during busy network periods for short processing latency. The adaption speed and stability are determined by its control policy, such as slow adaption, fast adaption or block adaption. Thus the adaptive Bloom filter maintains a dynamic balance of power and processing performance accordingly. Experiments show that EABF can achieve close to the best power savings and almost 1 clock cycle latency similar to that of a regular Bloom filter even in busy network conditions.

Thirdly, the two-stage EABF platform can be extended to a multi-stage energy-aware adaptive Bloom filter, which presents better suitability to network characteristics.

Finally, a cache-based CBF system on FPGA is proposed for higher performance and precise matching. The traditional Bloom filter has two main drawbacks: firstly, it does not support online update; second, it produces false positives. In order to reduce the number of off-chip memory accesses, a compressed CBF array is designed and pattern grouping is used for cache replacement. Considering pattern relativity and traffic locality, patterns are categorized in groups. For a potential match, it first checks the cache index to see if hash entries exist in cache; if not, it searches off-chip

memory and replaces a cache-line with a pattern group. Experiments show that C^2BF can significantly reduce match verification time. Several cache schemes and cache replacement policies are also compared under different traffic conditions. The cache hit rate can reach more than 80%, which reduces more than 70% of matching verification time compared to the traditional schemes without cache.

Chapter 5: Future Work Proposal

- Power Modeling and Low Power DPI

This chapter is the proposed future work of models and methods for low power DPI. Except for the preliminary results in Section 5.2, other results for the work in this chapter are not available since they are still limited to the models and designs of the algorithms, the application of which is my future work on low power DPI.

This chapter proposes two power saving models for routers and DPI respectively, which can be used to evaluate the power saving contribution and the latency impact of dynamic frequency adaption to the traffic with two or more frequencies. Another model is proposed to illustrate the fluctuation problem in frequency adaption, and the repeated fluctuation of running frequencies can be alleviated by the use of two thresholds as a region, among which the frequency stays the same. The low power designs for one or multiple DPI engines are also proposed.

5.1 Introduction

In recent years, statistics of network energy requirements and the related carbon footprint show an alarming and growing trend as reported by ISPs. It is estimated that 5.3% of global energy consumption is spent on Internet related devices. Based on the statistics in [6], the Global e-Sustainability Initiative (GeSI) estimates an overall network energy requirement of about 21.4 *TWh* in 2010 for European Telcos, and foresees a figure of 35.8 *TWh* in 2020 without low power techniques. One reason for the high energy consumption of the Internet is that the networking devices are originally designed to work at the maximum capacity regardless of the traffic load. The development of green network technologies provides network processing the capability of adapting its frequency for performance and energy efficiency [6].

IEEE 802.3az develops Energy Efficient Ethernet (EEE) standards in 2010 that allows for less power consumption during periods of low data activity [88]. The objective of EEE is to reduce power consumption by 50% or more, while retaining full compatibility with existing equipment. The potential savings of EEE standards are prominent. According to research in [89], the use of EEE could save an estimated $450 million a year in energy costs in the U.S., most of savings come from homes ($200 million), offices ($179 million) and data center ($80 million). Network device vendors have designed new products with power adaption following the EEE standards [88]. For instance, HP E5400, E5412, E8212 etc. EEE-enabled switching products [90] achieve more than 20% of power savings compared with early devices with similar capacity. A number of Ethernet related devices in offices are only used during business hours and can be smartly powered off when they not in use. But the smart standby of network devices requires the support of new protocols to save the physical power consumption.

Typically, 60% of power consumption in networking equipment is associated with packet-processing silicon and packet-processing support silicon such as memories (DRAM and TCAM's) [88]. To increase power savings, the best place to start is to reduce the power consumption associated with packet processing like DPI.

This chapter proposes the models and designs for power efficiency of DPI, including 1) the modeling of the router with dynamic frequencies, 2) the modeling of frequency adaption, 3) the modeling of the low power DPI, 4) the low power design of DPI with one and with multiple engines, respectively.

Current work of this chapter is based on power saving algorithms and models. However, due to the platform and time limitation, their results are not available in this thesis and the completion of these efforts is my future work on low power network design. In future work, the 10G-NetFPGA board with Xilinx Virtex-5 will be chosen as a target platform; based on the premise of basic speed requirement, my focus will be the power reduction of the DPI system. The architecture of this FPGA enables a smooth trade-off between speed and power, e.g. during the compilation and synthesis

procedure, a high-speed or low-power mode can be supported.

5.2 Modeling of Router with Dynamic Frequencies

In this section, a model is proposed for a router with dynamic frequencies, in comparison to a fixed frequency, to evaluate the power saving contribution and the latency impact of dynamic frequency adaption to the traffic with two or more frequencies. The model is built using queuing theory based on an approximate simulation of the real traffic flows. This section presents the preliminary work on modeling and the evaluation of this model is my future work.

The modeling work for a router with dynamic frequencies includes four aspects: 1) ***modeling***: a model is built to simulate a router with dynamic frequency adaption using queuing theory; 2) ***algorithm***: dynamic frequency adaption algorithms are presented for larger power savings; 3) ***analysis***: potential power saving and latency impact are formulated with the model; 4) ***application***: the model can work for different kinds of routers including core router, edge router and home router.

1) Modeling

Based on the queuing theory, a model is proposed for packet processing in a dynamic frequency scaling router. The router is considered as a queue, the header packet (i.e. customer) is processed by service windows, and new packets join at the tail of the queue. The modeling considers three parameters including ***service window***, ***queue*** and ***customer*** for network processing at a router.

- ***service window***

Different frequency corresponds to different number of service windows. It is assumed that the router has the capability of handling peak traffic speed. The maximum capability of the router corresponds to the maximum number of service windows. On average, traffic speed at the router is comparatively low, which corresponds to a single service window. Larger traffic speed requires a higher

frequency, which corresponds to a larger number of service windows. When the current service windows are not capable of handling traffic load at the queue, one more window is started.

- ***queue***

The models are proposed using infinity queues. This is based on the assumption that the router buffer is designed to be large enough to store the queuing packets in normal condition. A global queue is assumed to infinite. On the other hand, a local queue for a service window is assumed to be finite, so that the waiting time for a packet to be processed can be controlled. A new service window is started when the local queues of existing service windows are full.

The global queue can be shared by all the service windows, or each window has its local queue. In the first case, the header of the queue, shared for all the service windows, is sent to the first vacant window. If the number of customers in the queue exceeds a threshold, a new window is added. A service window is closed when the buffer length is below the threshold. The threshold is determined by the tolerable maximum waiting time of a packet. In the second case, each window has its own queue. If all the current queues are full, a new window is started and the new packet is scheduled to a new queue. When the queue of a service window has been empty for a certain period of time, the service window is closed. In this case, when a new customer (i.e. packet in this model) arrives, a scheduler is needed to send the customer to the first available queue.

- ***customer***

The customer in this model is network packet. Up to now, there hasn’t been an accurate model to describe the distribution of network traffic. Poisson distribution is used by the early research on the modeling of network traffic [91]. In recent years, more complicated models with dynamic parameters are proposed. But the new models are difficult to work with the queuing theory. In this thesis, the arrival traffic follows Poisson distribution with parameter λ, and the lengths of inter-arrival times follows

the exponential distribution, with mean $1/\lambda$. The service rate is related with current traffic conditions. Basically, the processing in a router is only related with the packet header and the buffer stores a pointer to the packet and the processing time of a packet can be regarded a constant value τ. Thus, the service durations of the packets are independent random variables with a common general probability distribution, the mean service is $D(\tau)$.

- ***modeling***

Suppose the router buffer is designed to be large enough to store the incoming packets in normal conditions, and there are s available service windows, network processing at a router with dynamic frequencies can be regarded as a ***M/D/c*** queue [92] ($1 \le c \le s$). In this model, the arrival process of customers (i.e. packets) is a *Poisson* process (*M*) with rate λ, the service time of a customer (i.e. the processing of a packet in a router) is a constant *D*, and *c* identical servers are available during a given sample period. It is assumed that the server utilization ρ, $\rho=\lambda D/c$, is smaller than 1; when ρ gets close to 1, a new server is added and *c* is increased by 1.

The model starts from a fixed rate single window system as ***M/D/1*** queue and gradually adapts the number of service windows according to arrival rate λ and the number of customers at the queue (i.e. the traffic load at the buffer).

2) Algorithm

Frequency scaling algorithm normally works as follows. According to current traffic load, the running frequency is tuned higher or lower, which can save power and complete the packet processing. Suppose there are *k* available frequencies, from f_1 to f_k, corresponding to *k* states.

The frequency adaption can be based on three indicators: 1) ***buffer length***, which is the number of packets in the queue; 2) ***traffic speed***, which is the number of packets arrived during a unit period; and moreover, 3) ***traffic acceleration speed***, which is another indicator that describes the incremental trend of network traffic. The acceleration speed responds faster to traffic bursts.

Three methods of frequency scaling are proposed for different application scenarios: 1) ***smooth adaption***; 2) ***leap adaption***; 3) ***selective adaption***, as show in Figure 5-1.

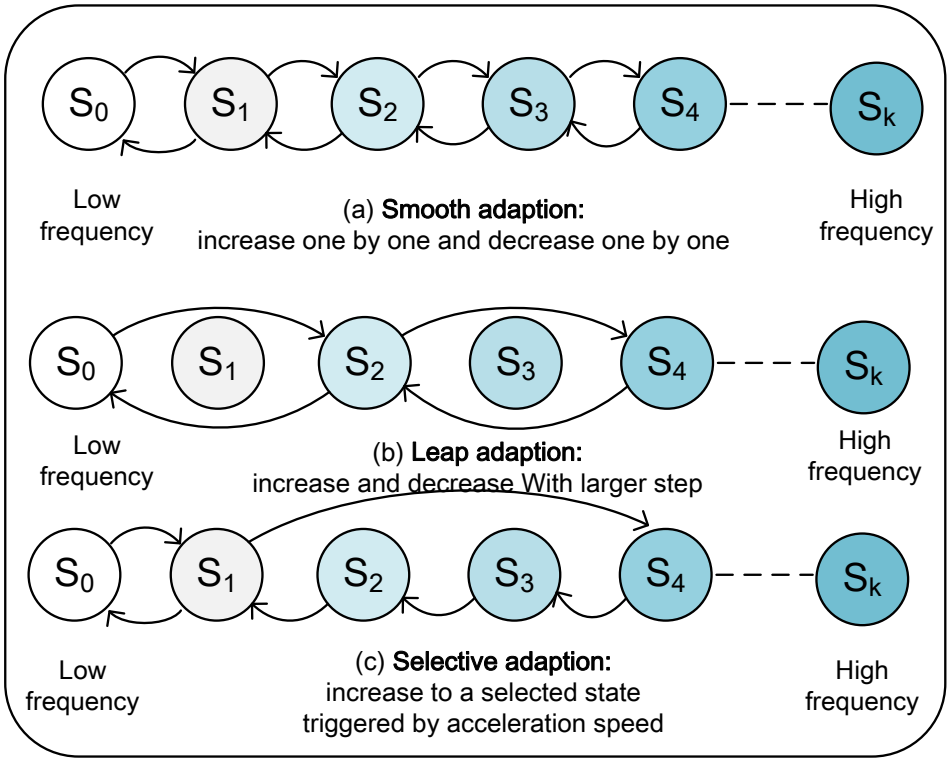


Figure 5-1 State transitions for frequencies with different policies

- **Smooth adaption to the neighboring state**

For stable traffic conditions, the smooth policy upgrades and degrades one by one, as shown in Figure 5-1 (a), and this method produces a lower packet loss rate.

- **Leap adaption with larger steps to the next neighboring state**

For traffic conditions that change fast from low speed to a high speed, the leap adaption goes to the next neighboring state with larger steps, as shown in Figure 5-1 (b). This method adapts faster compared to smooth adaption. This method is expected to achieve larger power savings, at the cost of the stability of router processing.

- **Adaption to a selected state by monitoring acceleration speed**

For great bursts in network traffic, the system can go to a selective state with a

required high frequency using selective adaption, as shown in Figure 5-1 (c). The traffic burst is monitored by traffic acceleration speed.

3) Analysis

Compared to the fixed number of service windows, the average number of active windows can be obtained for traffics with *Poisson* distribution. The total time (waiting time and service time) that a packet stays in a router is also presented. The parameters used in the model are explained in Table 5-1.

Table 5-1 Parameters in modeling for power savings

Variable	Meaning
s	The maximum number of service windows
c	Current number of service windows
μ	Service speed of a single window (Constant)
λ	traffic arrival rate (*Poisson*)
ρ	service intensity

Firstly, the model is a fixed rate single window system and the queuing system belongs to ***M/D/1*** model. The system parameters at stable states are presented as follows, which are used to constrain the minimum feasible frequency for an average traffic condition [92].

1) System idle probability is $P_0 = 1-\rho$;

2) The number of packets in the router is $L = \rho + \frac{\rho^2 + \lambda^2 D(\tau_n)}{2(1-\rho)}$;

3) The number of packets waiting in the buffer is $L_q = \frac{\rho^2 + \lambda^2 D(\tau_n)}{2(1-\rho)}$;

4) The time a packet stays and waits in the router are $W = L/\lambda, W_q = L_q/\lambda$.

Secondly, when current frequency is not capable of handling arrival traffic, the

system is dynamically transformed to a multi-server model, *M/D/s*, where *s* is the maximum number of windows. A larger frequency is related with a larger number of service windows, for example, $f_2=2f_1$ corresponds to two service windows, and each window has a minimum processing capability f_1. The frequency adaption corresponds to the active number of windows in service. If there are idle windows in the multi-server *M/D/s* system, the router frequency is reduced to a lower level. Suppose there are *c* active windows at present ($c \le s$), the average service rate of a single window is μ, the service rate for *c* windows is $c\mu$ if $n \ge s$, or $n\mu$ if $n<c$ where *n* is the number of packets in the system. Suppose the traffic arrival speed is λ, then the server utilization (i.e. service intensity) is $\rho = \lambda / c\mu$ and $\rho < 1$.The number of working windows is evaluated as follows [92].

1) the probability that there are *j* packets in the system is

$$P_j = e^{\lambda D}\frac{(\lambda D)^j}{j!}\sum_{k=0}^{c} P_k + \sum_{k=c+1}^{c+j} P_k e^{-\lambda D}\frac{(\lambda D)^{j-k+c}}{(j-k+c)!};$$

2) the probability that all windows are busy is $\sum_{j=c}^{\infty} P_j = 1-\sum_{k=0}^{c-1} P_k$;

3) the mean number of windows in service is $\sum_{j=0}^{c} j \cdot P_j + \sum_{j=c-1}^{\infty} c \cdot P_j$.

Finally, the queuing system can also be extended to a multi-queuing system as ***M^X/D/s*** in order to describe the model for a system with multiple processors, where the packets arrive in batches rather than in isolation. The arrival process of batches is a *Poisson* process with rate λ. The batch size has a probability distribution $\{\beta_j, j = 1,2,...\}$ with finite mean β. The service times of the customers are independent of each other and have a general distribution with mean *E*(*S*). There are *c* identical servers. It is assumed that the server utilization ρ is defined by $\rho = \lambda\beta E(S)/c$. The probability that there are *j* packets in the multi-router system is

$P_j = r_j(D)\sum_{k=0}^{c} P_k + \sum_{k=c+1}^{c+j} r_{j-k+c}(D)P_k$,where $r_j(D)$ is the probability that there are j packets arriving in the system during the sample period [92].

4) Application

This model can be applied to different kinds of routers deployed in the network: home router, edge router and core router. The majority of Internet energy is consumed by home routers, which are also called residential gateway, connects devices in the home to the Internet or other WAN. There are multiple devices like cable or DSL modem, wireless router or wireless access point, network switch etc. Edge router is placed at the edge of an ISP network, and it connects the network you control to a network that you don't. A core router resides within an Autonomous System as a back bone to carry traffic between edge routers. Core routers are optimized for high bandwidth. The parameters should be different for the deployment of dynamic frequency on different kinds of routers, considering their flow characteristics and requirements.

5.3 Modeling of Low Power DPI

The low power design for DPI with dynamic frequency scaling is modeled in this section. Method of energy efficient DPI has been investigated in academia but it is still lack of theoretical support. This evaluation of this model is my future work.

Similarly to the model in Section 5.2, low power DPI system can be modeled as a multi-server station at which network packets arrive according to a *Poisson* process with rate λ. There are s servers shared with an infinite buffer. If there is an available server for a new packet in the buffer, the packet is processed immediately; otherwise, the packet waits in the buffer. The service time of a packet is related with the packet length since DPI needs to process each byte in packet payload. Since there is no accurate description about packet size distribution [93], *Poisson* process is still a reasonable assumption when no other data is available about packet sizes [94]. The

service times of the packets are then independent random variables having a common exponential distribution with mean $1/\mu$.

This model is abbreviated as M/M/s queue and the service intensity of $\rho = \frac{\lambda}{s\mu}$ meets $\rho < 1$[92]. This is based on average rates, but instantaneous arrival rate may exceed the service rate. In view of a long period, the service rate should always exceed arrival rate.

Initially, it is assumed that no packets will be dropped and there is a single DPI engine working, and the model is simplified as a basic M/M/1 queue. This model is used to evaluate the traffic arriving rate that a single DPI engine is capable of handling, or the minimum required processing frequency for a single DPI engine. The parameters at a stable state are shown as follows.

1) The probability of *n* packets in the system is $p_n = P\{N = n\} = \rho^n p_0$;

where p_0 can be calculated as follows: $p_0 = \frac{1}{1 + \sum_{n=1}^{\infty} \rho^n} = 1 - \rho.$

Then $p_n = (1-\rho)\rho^n, n = 1, 2, ...$

2) Mean number of packets in the system (*N*) is

$$N = \sum_{n=0}^{\infty} n \cdot p_n = \rho / (1-\rho) = \lambda / (\mu - \lambda);$$

Mean number of packets in the buffer queue (*Q*) is

$$Q = \sum_{n=1}^{\infty} (n-1) \cdot p_n = N - (1 - p_0) = N - \rho = \lambda^2 / \mu(\mu - \lambda);$$

3) The total waiting time (including the service time) is $T = 1/(\mu - \lambda)$

It is also assumed that the processing frequency can be adjusted for two objectives: 1) the minimum required network throughput *v*, or the longest waiting time is $1/v$; 2) the average service intensity is close to 1, which means the maximum

utilization of DPI engine for power efficiency.

In view of the first objective that $T = 1/(\mu - \lambda) \le 1/v$ and the required frequency is $\mu_1 \ge \lambda + v$. In view of the second objective, suppose the system can hold at most N_{max} packets, for larger service intensity $\rho = N_{\max}/(1+N_{\max})$, the required frequency meets $\mu_2 \ge \lambda(1+1/N_{\max})$. As a result, the minimum frequency of a single DPI engine is $\mu_{\min} = \max\{\mu_1, \mu_2\}$. Similarly, based on a given frequency, the maximum arriving speed the DPI engine handles can be evaluated.

Moreover, the parallel DPI system is modeled as M/M/s queue, which is a straightforward extension of M/M/1 queue. The arriving traffic of rate λ is served with s DPI engines, each of which has a service rate μ. The service intensity with s windows is $\rho_s = \frac{\rho}{s} = \frac{\lambda}{s\mu}$, similarly, ρ_s should always be less than one. The parameters in stable state are shown as follows.

1) The probability of n packets in the system is

$$p_n = P\{N = n\} = \begin{cases} \frac{\rho^n}{n!} p_0, & n = 1, 2 \ldots s \\ \frac{\rho^n}{s! s^{n-s}} p_0, & n \ge s \end{cases};$$

where p_0 is calculated as follows: $p_0 = \left[\sum_{n=0}^{s-1} \frac{\rho^n}{n!} + \frac{\rho^s}{s!(1-\rho^s)} \right]^{-1}$.

2) The probability of waiting for a packet in this system is that the system when the number of packets in the system is larger than the number of windows s.

$$c(s, \rho) = \sum_{n=s}^{\infty} p_n = \frac{\rho^s}{s!(1-\rho^s)} p_0$$

3) Mean number of packets in the queue (Q) is as follows, the queuing starts when n equals to s+1.

$$Q = \sum_{n=s+1}^{\infty} (n-s) \cdot p_n = \frac{p_0 \rho^s}{s!} \sum_{n=s+1}^{\infty} (n-s) \cdot \rho_s^{n-s}$$

In the parallel DPI system, it is assumed that a new engine is started when the existing engines cannot handle the traffic fast enough in the buffer. In order to meet the maximum waiting time for a packet to be processed, each service window is associated with a finite local queue of length *K*, where one packet is processed in service window and the other maximum (*K*-1) waits in the queue. When the queue is full, the new packets are allocated to the local queue of Engine 2. In other words, Engine 2 is started when the waiting queue of the *M/M/1/K* system saturates at *K*. The probability of starting Engine 2 and the average queue length for Engine 1 is shown as follows.

1) Considering the queue for window 1, the probability of n packets in the sub-system for window 1 is $p_n = \rho^n p_0, n = 1, 2...K$.

where p_0 is calculated as: $p_0 = \frac{1}{1+\sum_{n=1}^{K}\rho^n} = \begin{cases} \frac{1-\rho}{1-\rho^{K+1}}, & \rho \neq 1 \\ \frac{1}{K+1}, & \rho = 1 \end{cases}$

It is possible for $\rho = 1$ that the queue length of window 1 system stays constant.

2) The probability of starting window 2 is calculated as $p_K = \rho^K p_0$.

3) After more windows are started, the average queuing length of window 1 system is shown as follows, and maximum queue length is *K*-1.

$$L_q = \begin{cases} \frac{\rho}{1-\rho} - \frac{1-\rho}{1-\rho^{K+1}}, & \rho \neq 1 \\ \frac{K(K-1)}{2(K+1)}, & \rho = 1 \end{cases} .$$

Similarly, Engine (*s*+1) is powered on when the existing *s* engines are not capable of processing the traffic. The probability of starting Engine (*s*+1) is P_K as follows [92].

$$p_K = \frac{\rho^K}{s!s^{K-s}} p_0 = \begin{cases} \frac{\rho^K}{s!s^{K-s}} \left(\sum_{n=0}^{s-1} \frac{\rho^n}{n!} + \frac{\rho^s (1-\rho_s^{K-s+1})}{s!(1-\rho_s)} \right)^{-1}, & \rho_s \neq 1 \\ \frac{\rho^K}{s!s^{K-s}} \left(\sum_{n=0}^{s-1} \frac{\rho^n}{n!} + \frac{\rho^s (K-s+1)}{s!} \right)^{-1}, & \rho_s = 1 \end{cases}.$$

5.4 Modeling of Fluctuation in Frequency Adaption

For dynamic frequency scaling, threshold in buffer length determines the number of times for frequency adaption. This section presents the modeling for frequency fluctuation based on a single threshold or two thresholds and shows the initial results for comparison, more experiments are needed in future work.

- **Problem Statement**

Although dynamic frequency scheme saves much power, there is a serious problem of fluctuation among modes with different frequencies, which leads to large overhead but has been neglected by previous researches. Suppose the system has fast and slow modes with running frequencies f_1 and f_2, divided by a threshold T, the frequency adaption is defined in Figure 5-2. For simplicity, considering the arriving traffic with a constant average speed, under certain conditions, the working mode will swiftly switch among the two modes. For example, suppose the buffer changes from l_n to l_{n+1} after one sample period, the average arrival load is Δl and the processed load is Δu with the low frequency if $l_n<T$, or $2\Delta u$ with the high frequency if $l_{n+1}>T$. If ($\Delta u<\Delta l<2\Delta u$), l_n and l_{n+1} will always move up and down along the threshold T, which causes oscillation in frequency adaption.

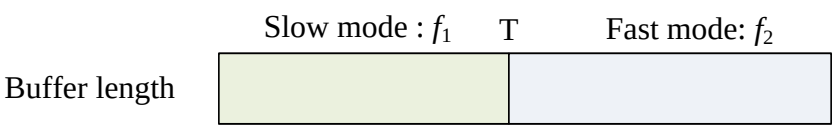


Figure 5-2 Frequency scaling with one threshold

- **Solution**

As a solution, instead of one threshold, a threshold region is specified with a

upper bound T_H and a lower bound T_L, between which the frequency remains unchanged. The frequency adaption is defined in Figure 5-3. For the above example, if the threshold region meets the condition that (T_H-T_L>max{2Δu-Δl, Δl-Δu}), the mode oscillation can be alleviated, since the threshold region provides a buffer zone.

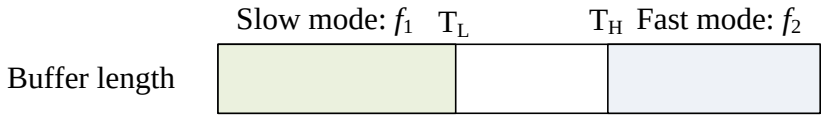


Figure 5-3 Frequency scaling with two thresholds

- **Experimental Comparison**

Three situations of network traffic are developed for experimental comparison of frequency fluctuation based on one or two thresholds: 1) *constant traffic*; 2) *random traffic following Poisson distribution*; 3) *real traffic extracted from a trace at edge router*. From the experiment, the use of two thresholds for frequency adaption is much smoother than that of a single threshold, which can reduce adaption overhead and improve the performance of dynamic frequencies.

1. Experiment setup

The model is implemented on a simulated router written in Python and uses the parameters in Table 5-2 and illustrated in Figure 5-4. In the experiment, we compare the frequency fluctuation using two or three frequencies. Testcases are designed for the following situations: 1) below the low frequency; 2) between two running frequencies and 3) above the high frequency. Theoretically, the first group will work stable at the lowest frequency, the second group will fluctuate between two frequencies, and the third group will work stable at the high frequency.

In addition, the frequencies are set as the number of clock cycles, e.g. 125MHz is converted to 8*ns* as a clock cycle and 62.5MHz is converted to 16*ns* as a clock cycle. The basic processing of a packet requires 6 clock cycles on FPGA, in view of 8*ns* as a clock cycle, it takes 6 clock cycles using 125MHz or 12 clock cycles using 62.5MHz. Similarly, the packet arrival speed is also set in units of packet intervals, e.g. the short

interval of 4 clock cycles corresponds to the fast speed of 187.5MHz and the long interval of 15 clock cycles corresponds to the slow speed of 50MHz etc.

Table 5-2 Parameters in frequency fluctuation modeling experiment

Variable	Meaning	Values in the experiment
Δu	The number of clock cycles required to process a packet	6, 12, 18 for three levels of frequencies
Δl	The average number of clock cycles between two packets	4, 9, 15, 20 for four levels of speeds in testcases
L	buffer length	300
S	Sample periods	48, 72, 96
T	Threshold for a single threshold scheme	150 for two frequencies; 100, 200 for three frequencies
T_H	High threshold of a threshold region	180 for two frequencies; 120, 220 for three frequencies
T_L	Low threshold of a threshold region	120 for two frequencies; 80, 180 for three frequencies
B	Alert line for fine-grained adaption	280

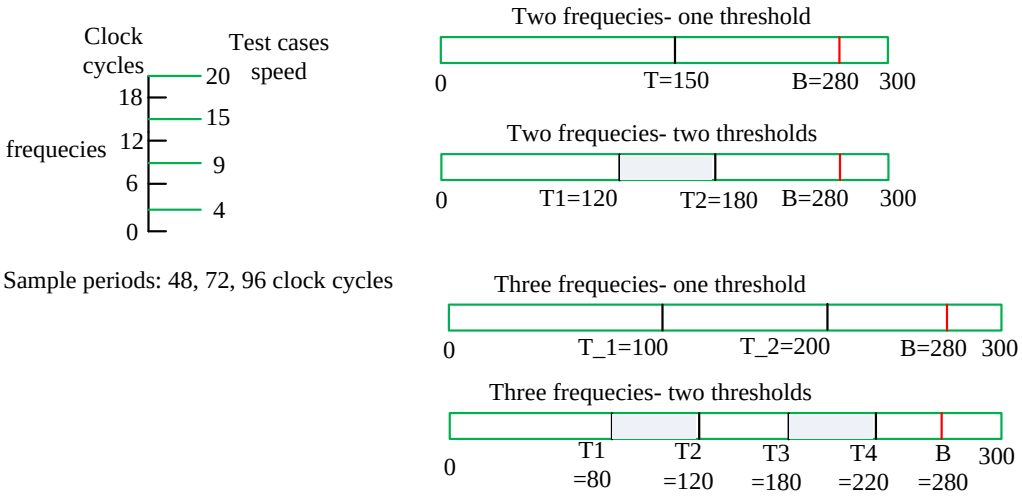


Figure 5-4 Parameters in frequency fluctuation modeling experiment

2. Experiment result

The preliminary experiment results with constant traffic are shown below. Figure 5-5 shows the running frequency for constant traffic with one threshold, where f_L

stands for low frequency and f_H stands for high frequency. Three groups of packet intervals are compared. Firstly, when the constant packet interval is “4” or “15” clock cycles, which are below or above two frequencies with clock periods of “6” and “12” clock cycles, the frequencies are stable at f_L or f_H, with no more than one frequency adaption. Secondly, when the constant packet interval is “9”, which is between two frequencies with clock periods of “6” and “12” clock cycles, frequency fluctuation appears with 80 times of adaption within 200 clock cycles. Accordingly, as shown in Figure 5-6, the buffer length changes up and down 200 times around the threshold of “150”.

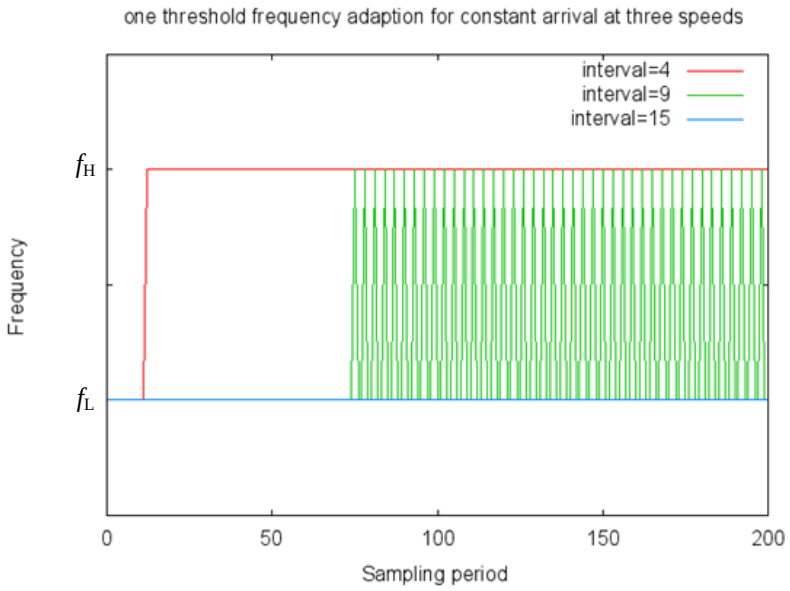


Figure 5-5 Running frequency of constant traffic with one threshold

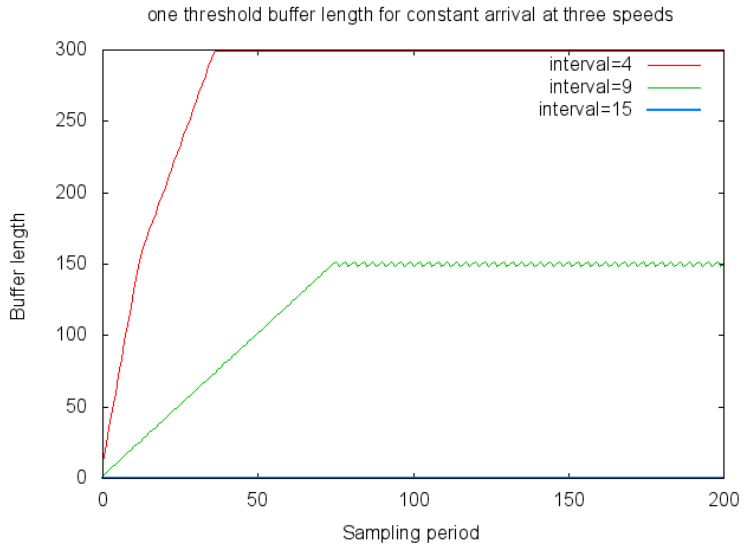


Figure 5-6 Buffer length of constant traffic with one threshold

The fluctuation can be relieved by the use of two thresholds, among which the frequency does not change. Figure 5-7 shows the running frequency of constant traffic with two thresholds, where f_L stands for low frequency and f_H stands for high frequency. With packet intervals of “4” or “15” clock cycles, which are below or above two frequencies with clock periods of “6” and “12” clock cycles, the frequencies are the same as that of a single threshold. With packet intervals of “9” clock cycles, which is between two frequencies “6” and “12”, frequency fluctuation appears but is much smoother than that of a single threshold. Accordingly, as shown in Figure 5-8, the buffer length changes up and down only 5 times around the threshold of “150”.

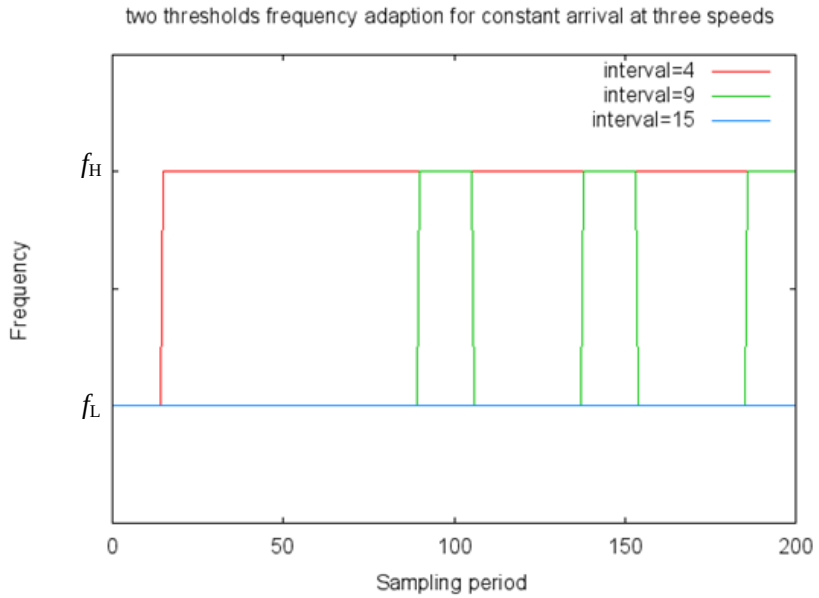


Figure 5-7 Running frequency of constant traffic with two thresholds

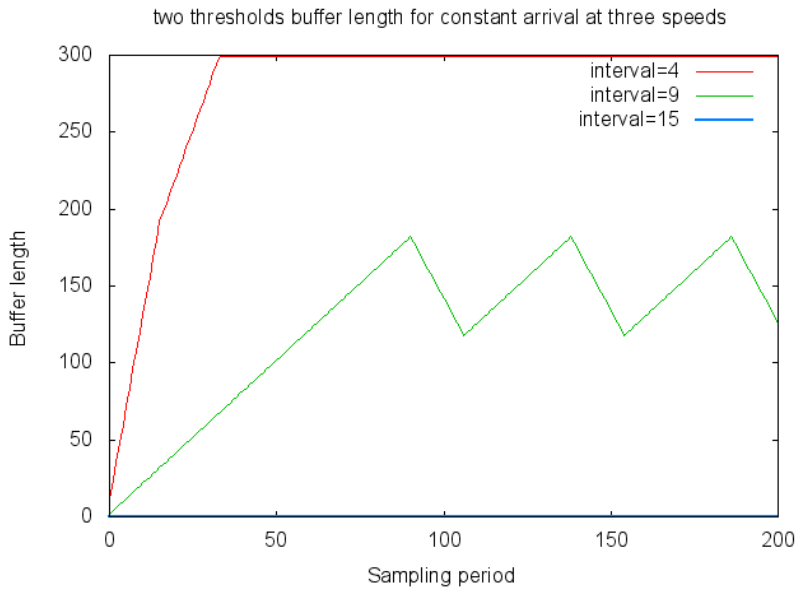


Figure 5-8 Buffer length of constant traffic with two thresholds

For comparison with one or two thresholds, Figure 5-9 shows frequency fluctuation for two frequencies and Figure 5-10 shows the buffer length when it reaches around the threshold.

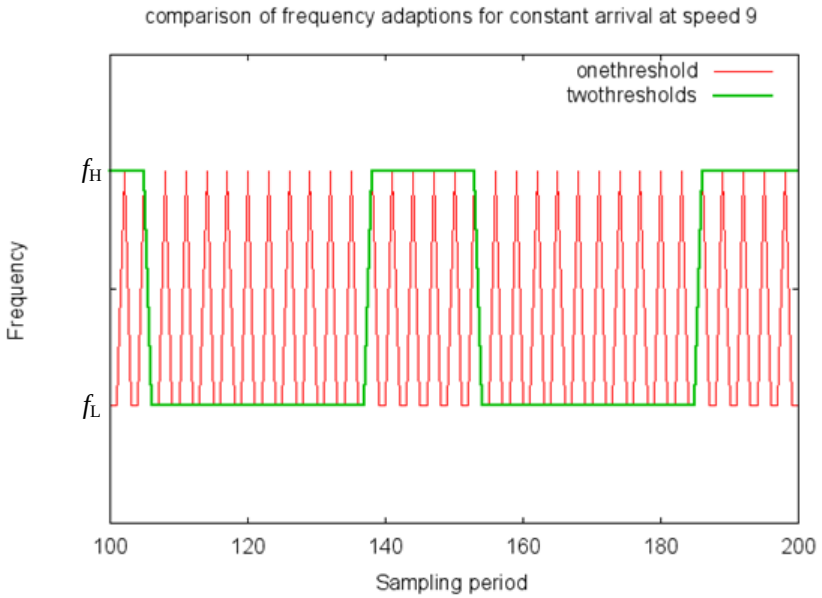


Figure 5-9 Comparison of frequency with one or two thresholds

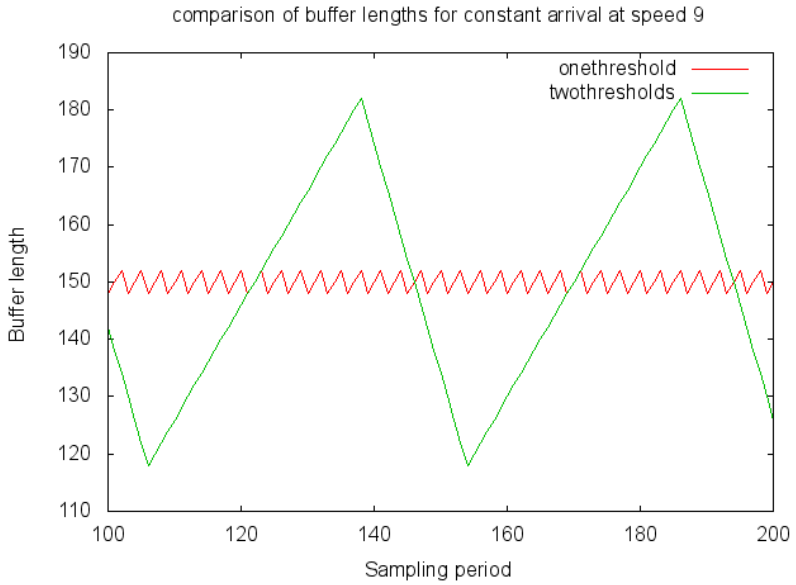


Figure 5-10 Comparison of buffer length with one or two thresholds

In simulation of the *Poisson* traffic, three groups of packet intervals are generated as random numbers following negative exponential distribution with the mean values of “4”, “9” or “15” clock cycles.

As shown in Figure 5-11, when the average packet interval is “9” clock cycles, which is between two frequencies with “6” and “12” clock cycles, frequency

fluctuation appears after the buffer length is close to the threshold. When the average packet intervals are“4” or “15” clock cycles, which are below or above two frequencies with “6” and “12” clock cycles, the frequencies stay stable at f_L or f_H, with no more than one frequency adaption. Figure 5-12 shows the buffer length with *Poisson* traffic.

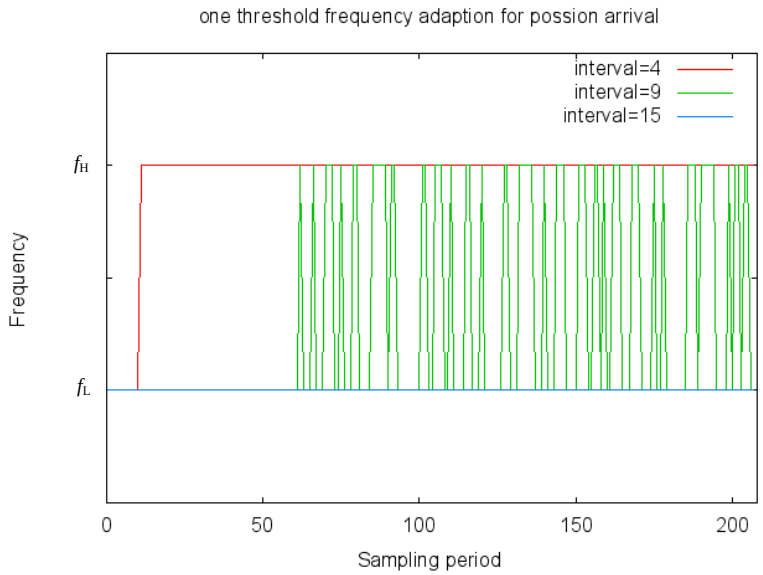


Figure 5-11 Running frequency of *Poisson* traffic with one threshold

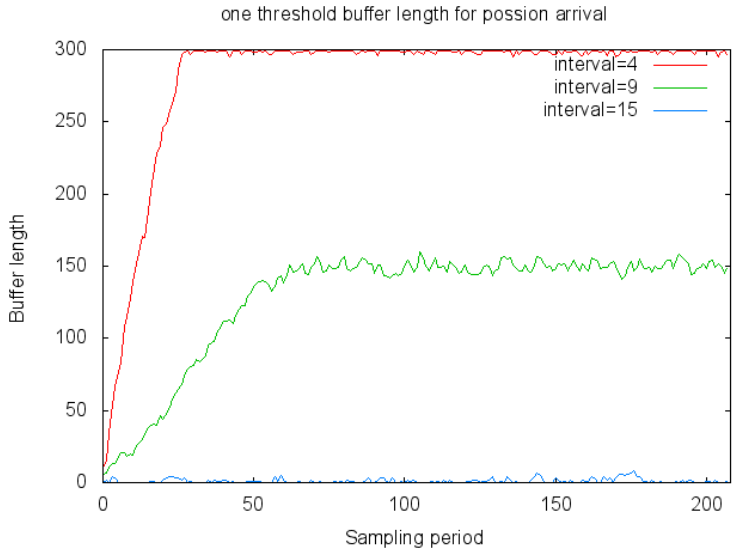


Figure 5-12 Buffer length of *Poisson* traffic with one threshold

The fluctuation can be relieved by the use of two thresholds, as shown in Figure 5-13 for the running frequency and Figure 5-14 for the buffer length of *Poisson* traffic

with two thresholds. Similarly to the constant traffic, with packet interval of "9" clock cycles, frequency and buffer length fluctuation appears but is much smoother than that of a single threshold. In comparison of frequency adaption with one or two thresholds, Figure 5-15 and Figure 5-16 shows the comparison of frequency and buffer length fluctuation.

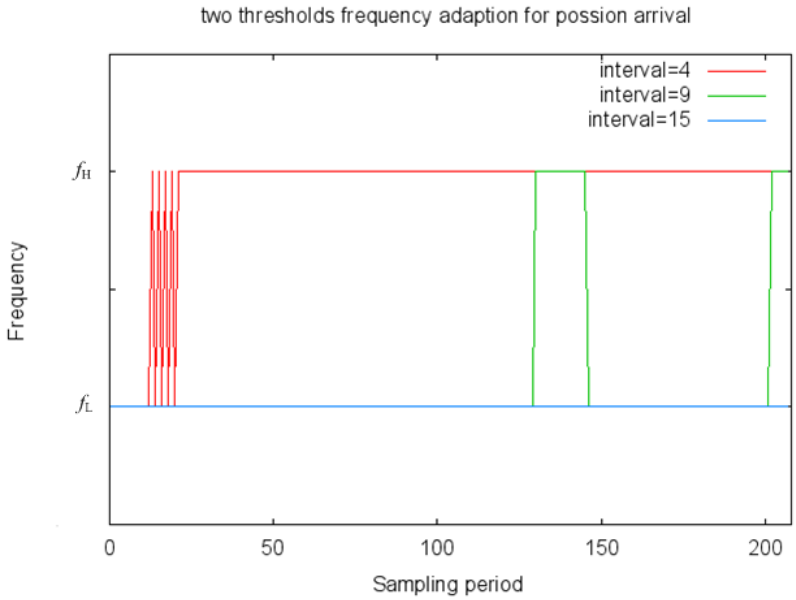


Figure 5-13 Running frequency of *Poisson* traffic with two thresholds

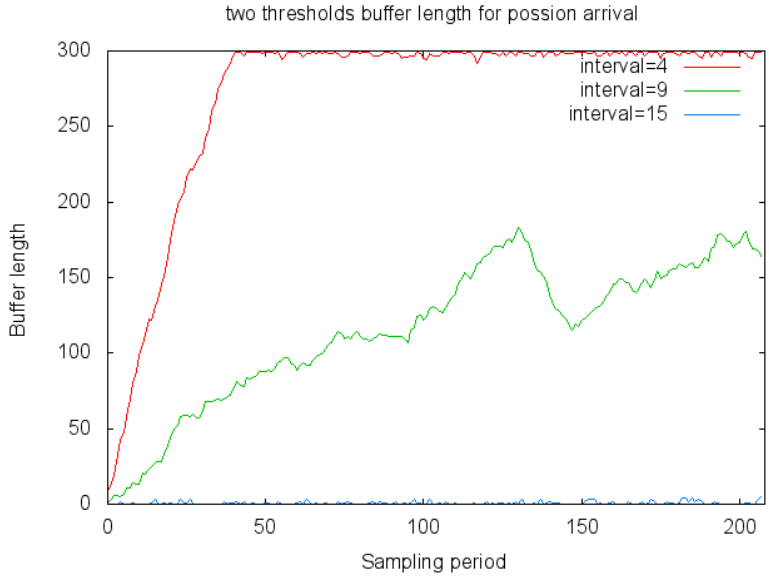


Figure 5-14 Buffer length of *Poisson* traffic with two thresholds

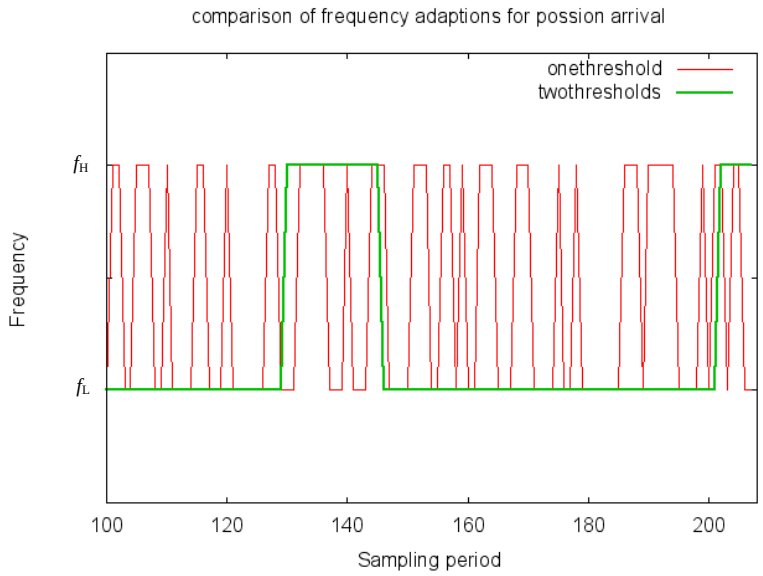


Figure 5-15 Comparison of *Poisson* frequency with one or two thresholds

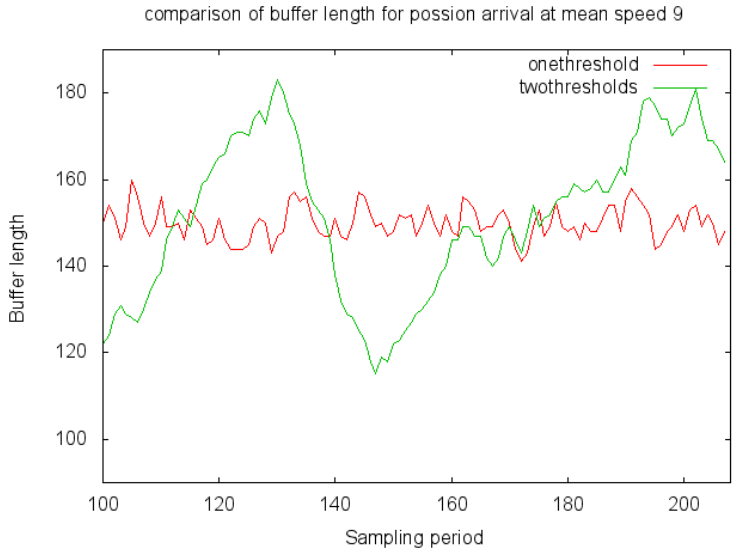


Figure 5-16 Comparison of *Poisson* buffer length with one or two thresholds

5.5 Single Engine Power Control

This section proposes the power control method for a single DPI engine, the evaluation and implementation of which is the future work.

5.5.1 Power Reduction Methods

At present, feasible approaches to reduce power consumption of a router can be

classified as dynamic power scaling and smart standby [95][96]. Sleeping and standby approaches allow the network devices or parts of them to turn themselves almost completely off, and entering very low energy states, when all their functionalities are frozen [6]. Network devices cannot completely sleep since they need to maintain the "network presence", or they will become unreachable and fall off the network. Dynamic frequencies are used in this chapter to reduce power consumption of a router.

On the other hand, a sleep mode can be used in the low power DPI system, since DPI is not required to be always active. The sleep mode brings significant energy savings by maintaining a minimum number of components to be active to handle the pattern matching processing. Previous work normally employs dynamic frequency control by observing buffer occupation, thus the response time is slower than that of optimal solution. When idle period is finished, the system has some unavoidable delays before returning running state.

Traditionally, FPGA applications run at a fixed frequency determined through static analysis in tools from FPGA vendors. The running frequency is constrained by its critical data path that produces the maximum delay. However, such a clocking strategy cannot take advantage of the full run-time potential of an application running on a specific device and in a specific operating environment. The dynamic clock and voltage methods are capable of overcoming this limitation. As alternatives to static estimates, dynamic frequency schemes have the potential to meet both high-performance and low-power objectives.

Austin et al. in [97] investigate opportunities for better than worst case hardware design for computers, through the scaling of voltage or clock during the worst-case inputs. Similarly, as network traffic fluctuates from time to time, the idea of a design for the average case performance is to create a system which can be adapted to some higher frequencies when inputs are above average case. Schemes for adapting clock-frequency to specific processing capability have also been developed in [98] with clock period altered each cycle depending on which units are currently active.

High-performance applications can run at the maximum physically attainable speed, while low-power applications can in general manage power-consumption by optimally balancing clock-frequency, voltage supply, and the processing demand. Comparatively, voltage decrease is more effective to power reduction while frequency scaling is more widely supported by FPGA. Another strategy is to schedule the units that consume a high amount of energy at lower frequencies, so that these units can be operated at lower frequencies to reduce their energy consumption. Meanwhile, the low energy units at higher frequencies, to compensate for speed. For example, suppose a number of units form a serial link to complete a complex calculation, the high-energy multipliers can operate at lower frequencies, the decreased speed is compensated by the low-energy adders at higher frequencies, and the total energy consumption is reduced.

Figure 5-17 (a) and Figure 5-17 (b) show the single and dynamic frequency schemes. Instead of a unique clock cycle, the dynamic scheme adapts clock cycles with its workload. Based on input frequency, which is usually the maximum system frequency, frequency divider or phase modulator for clock adjustment and alignment. The dynamic frequency scheme does not use a unified clock cycle. As shown in Figure 5-17(c), through software or hardware control, the output frequency f_{out} is d times smaller than f_{max}, and the phase is moved m degrees to align with reference clock cycles.

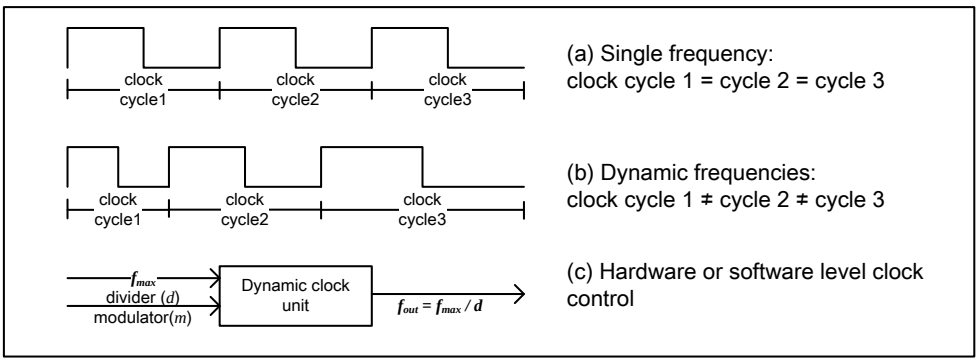


Figure 5-17 Singe and dynamic frequency schemes

5.5.2 Power Reduction with Dynamic Frequencies

Figure 5-18 is an illustration of adaptive clocking system, where the adjustable clock unit works as a module for a hardware accelerator. The adaptive clocking unit dynamically changes the clock of a processing engine (PE) to match fluctuations in traffic on a router. It needs a register to store the threshold values required to determine the running clock frequency. The clock control logic works by tentative probing within the upper and lower bounds. The packets are temporarily buffered in FIFOs.

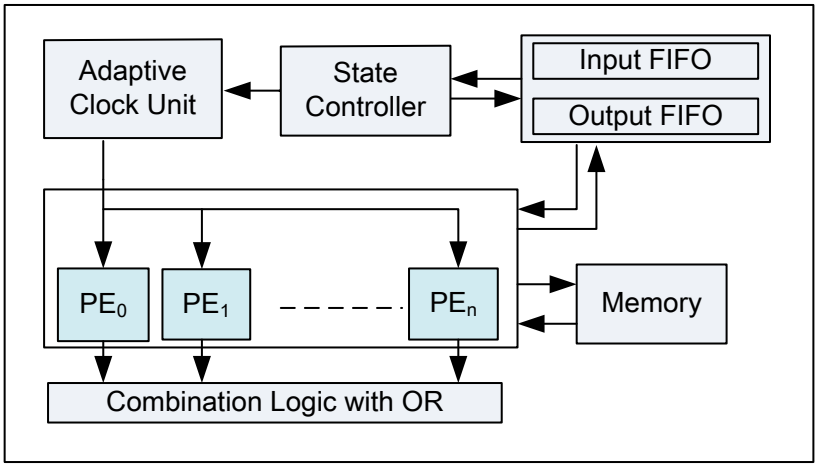


Figure 5-18 Structure for power control unit

The adaptive clock unit provides PEs with different frequencies according to the signals from State Controller. Moreover, one or more PE should be turned off when the whole workload can be handled with fewer active PEs in the system. The State Controller makes decisions based on the traffic condition and the working status of PEs. The traffic condition can be observed as the average arrival speed during a sample period, or the instantaneous acceleration speed for fast response. For example, when the average speed is normal, a short time burst can be captured by the change of acceleration speed; in this way, the frequency can be adjusted immediately.

The next question is how to determine the working status of a PE, there are three indicators that can be referred to in the algorithm, including: 1) the idle time of a PE

when the PE does no effective work, 2) the length of the thread queue in which a thread waits for incoming packets, and 3) the fullness of an internal packet buffer where packets come in and wait to be processed.

Moreover, a threshold to turn on or off PEs can be determined either statically with a fixed value obtained from off-line statistic data, or dynamically adjusted in hardware according to runtime information. The fixed frequency adaption threshold cannot satisfy all kinds of traffic conditions. In dynamic scheme, depending on the arrival traffic, the threshold value can be decreased or increased by a predefined value to see if the frequency adjustment has obvious performance impact or if there is a larger opportunity for power savings.

Similar to the adaptive clocking scheme in [99], a state machine whose states is associated with different running frequencies is built by setting a threshold, which is referred to as the number of packets in the input FIFO, to indicate current load condition in system. The system can be tuned to a lower state if the current load is below a threshold; in other words, the processing of current traffic load can be supported with a lower frequency.

According to the tune strategy, when the traffic load is below the threshold of the current state and the system can hardly provide the required throughput, it can be upgraded to the next higher state or a selected higher state; in the meantime, when the traffic load is below the threshold of the current state, it can be degraded to the initial state to test from the beginning for its condition, or it can go directly to a selected state to test from the medium stage, from where it could be adjusted to another higher or lower state again.

5.5.3 Threshold Determination

In dynamic power control schemes, power and performance is a tradeoff. The frequency switching process has an overhead. Consequently, power control unit would definitely have an impact on system performance. Luo et al. in [75] show that

the adjustment of voltage or frequency leads to unaffordable long latencies for high speed NPs.

The power control model adapts with two objectives: 1) it can handle the whole traffic load at a required throughput; 2) it has the minimum accumulative frequencies.

The following model is used for adaption illustration. Suppose f_{min} is the overall system frequency to meet the required minimum throughput, the average system frequency can be simplified as the sum of parallel engines. The system assigns a separate frequency for each engine using frequency and phase scaling. Suppose there are N homogeneous parallel Bloom filter engines, f_i is the frequency on engine i, then $(f_1, f_2, \ldots, f_N)$ should satisfy $f_1 + f_2 + \ldots, + f_N \geq f_{min}$.

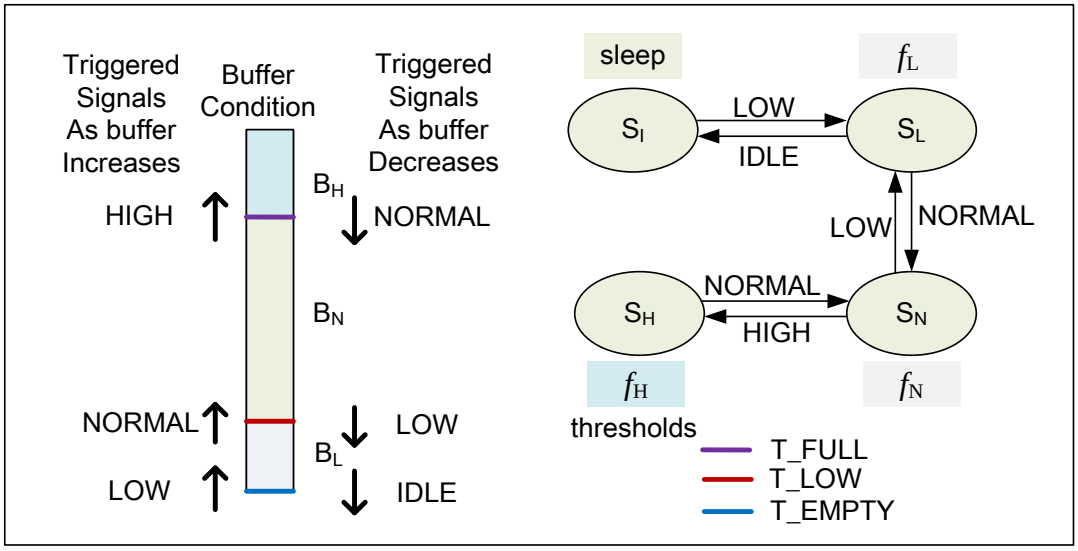


Figure 5-19 State machine and signals with buffer conditions

As shown in Figure 5-19, based on two thresholds, T_FULL and T_LOW, the buffer is divided three regions of B_H, B_N, B_L for high, normal and low conditions, respectively. Another threshold T_EMPTY is used for empty condition. Accordingly, a state machine is built with four states for power efficiency: 1) S_I for sleeping when buffer is empty; 2) S_L for running at a lower frequency f_L when buffer length is within the range B_L; 3) S_N for running at a normal frequency f_N when buffer length is within the range B_N; and 4) S_H for running at a higher frequency f_H when buffer length is within the range B_H.

In order to reduce the overhead of state switching, instead of checking buffer condition each clock cycle, four triggered signals are used: HIGH, NORMAL, LOW and IDLE. Table 5-3 shows the truth table of buffer thresholds. For example, when buffer increases from B_L to B_N, T_LOW changes from 1 to 0, and signal LOW is asserted by its negedge. In summary, signal IDLE is asserted by the posedge of T_EMPTY; signal LOW is asserted by the negedge of T_EMPTY or the posedge of T_LOW; signal NORMAL is asserted by the negedge of T_LOW or the posedge of T_NORMAL; and HIGH is asserted by the posedge of T_HIGH.

Table 5-3 Truth table of buffer signals

Buffer		0	B_L	B_N	B_H
States		S_I	S_L	S_N	S_H
thresholds	T_EMPTY	1	0	0	0
	T_LOW	0	1	0	0
	T_FULL	0	0	0	1

The determination of a threshold can be categorized into three methods: 1) fixed division; 2) statistical fixed division and 3) adapted division to workload.

Firstly, the thresholds can be chosen as the fixed divisions of buffers. For example, the 80% of a buffer length can be regarded as the high threshold and the 20% of the buffer length can be regarded as the low threshold. The adaption can be immediately triggered or it can wait to see if it stays above or below the threshold for some time. The delay trigger can effectively solve the frequent fluctuation between the high and the low frequencies.

Secondly, the thresholds can be obtained as empirical values from statistics of traffic loads. The fixed method is easy but rough, it does not consider network traffic conditions. Network traffic has self-similarity (that is the whole has the same shape as one or more of parts), and the long term traffic distribution also works for the arriving traffic to a buffer during short term period. Early traffic model uses Poisson distribution to describe Internet traffic fluctuations, and this model also applies to the arriving traffic load at the buffer. Take the buffer as a queuing system, the arriving traffic follows Poisson distribution, the service window is the processing engine and

the leaving traffic is related with the processing speed. Then the average number of packets waiting in the queue can be regarded as the buffer threshold.

Thirdly, the thresholds can be dynamic values that are modified periodically in consistent with current traffic conditions. For instance, a simple criterion can be roughly *day-threshold* for more occupied buffer and *night-threshold* for less occupied buffer. Compared to pre-determined fixed values, dynamic thresholds show better flexibility to real-time buffer conditions. A statistical module records the number of packets and their waiting time during this period, if there are too many packets in the buffer or if the waiting time is too long, the threshold should be adjusted accordingly.

5.5.4 Power Control Scheme

For simplicity, the power control scheme for low power DPI is explained with two frequencies including the high frequency, f_H, and the low frequency, f_L.

As shown in Figure 5-20, the power switching system adapts among three states, $\boldsymbol{S_H}$ for f_L, $\boldsymbol{S_L}$ for f_L, and $\boldsymbol{S_I}$ for sleep state that saves dynamic power. On the other hand, waking up from $\boldsymbol{S_I}$ takes a longer time than frequency adaption time between $\boldsymbol{S_H}$ and $\boldsymbol{S_L}$.

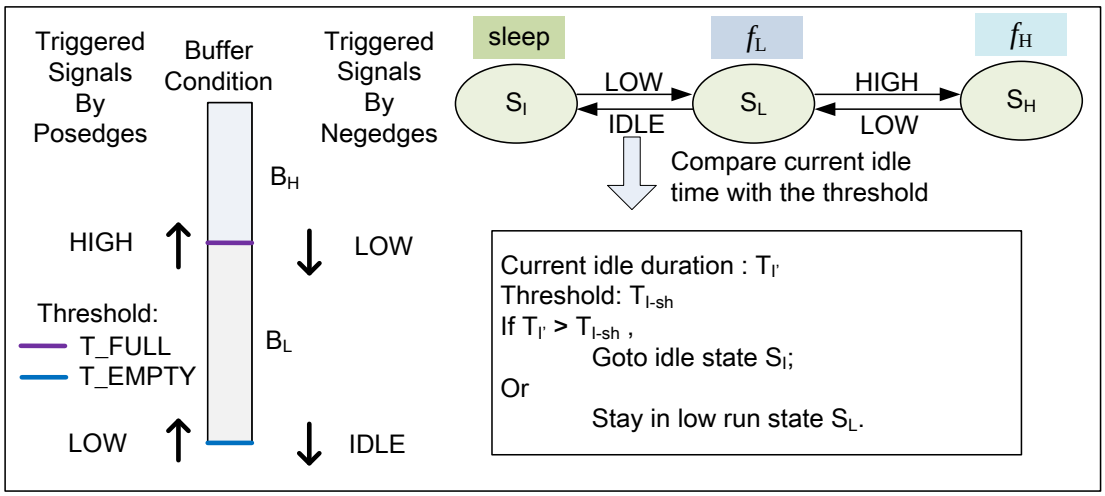


Figure 5-20 Simplified state machine and signals with buffer conditions

In addition, resuming from the sleep state consumes more power than the running power; therefore, it is an important decision to sleep or not, depending on the

incoming idle time. Previous work normally employs dynamic frequency control by observing buffer occupation, and the response time is badly affected due to the unavoidable delay before resuming to the running state. It is not worth to sleep down if the idle period is too short.

Consequently, the system has to wait a period before turning to the sleep mode. The question is how long it should wait before turning a PE to the sleep mode, that the total power consumption can be reduced. Based on traffic conditions, let T_{BH} be the time period during B_H, T_{BL} be the time period during B_L, T_{BI} be the idle period with empty buffer. Based on state machine, T_H, T_L and T_I are durations of states S_H, S_L and S_I, T_E is the switching time from the running state S_L to the sleep state S_I, and T_W is the delay overhead for waking up from the sleep state S_I to the running state S_L. The switching time between high and low frequency time can be relatively neglected.

In Figure 5-21, the working status of the PE is illustrated in line with the arriving traffic workload. Suppose T_{BI} is based on buffer conditions, T_I is the idle time and $T_{I'}$ is the waited idle time before turning to the sleep mode after IDLE is triggered. From Figure 5-21 (b), idle time T_I should be longer than the sum of T_E and T_W. We analyze the threshold $T_{I\text{-}sh}$ of idle time for low power.

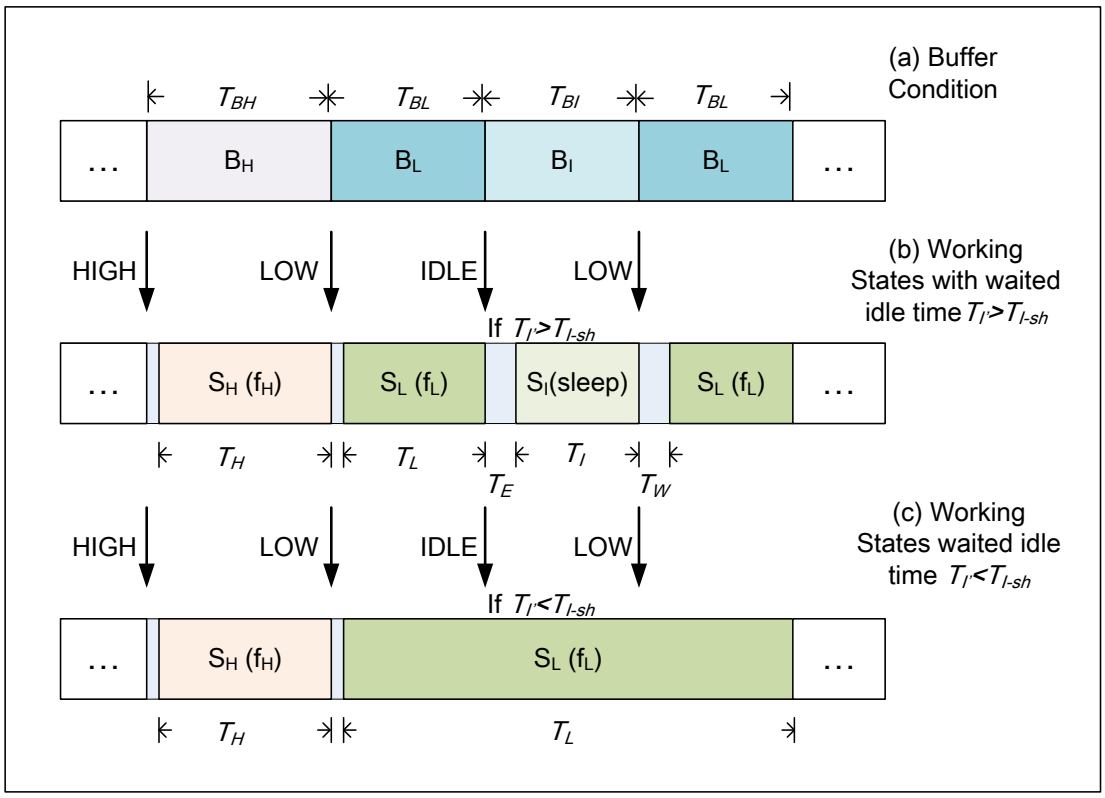


Figure 5-21 Buffer conditions and working states

Suppose P_{HR} is the power with high frequency, P_{LR} is the power for low frequency, P_I is the sleep power and P_T is the transition time power when entering or leaving idle state. Let E_D denote the energy difference of the system with or without power control unit, suppose $T_{BI} \geq T_E + T_W$ and $P_T \geq P_I$, the energy difference E_D is shown in Equation (5-1).

$$\begin{aligned} E_D &= P_{LR} \cdot T_{BI} - P_I \cdot T_I - P_T \cdot (T_E + T_W) \\ &= P_{LR} \cdot T_{BI} - P_I \cdot (T_{BI} - T_E) - P_T \cdot (T_E + T_W) \\ &= (P_{LR} - P_I) \cdot T_{BI} - (P_T - P_I) \cdot T_E - P_T \cdot T_W \end{aligned} \tag{5-1}$$

For $E_D > 0$, threshold idle time $T_{I\text{-}sh}$ is calculated as Equation (5-2).

$$T_{I-sh} = T_{BI} > \frac{P_T \cdot T_W + (P_T - P_I) \cdot T_E}{P_{LR} - P_I} \tag{5-2}$$

The power control scheme for low power DPI is based on the states in Figure 5-22. The DPI system initially works at S_L state. When IDLE is triggered, it waits an

idle time $T_{I'}$ when the buffer stays empty. If the idle signal is disabled before it reaches the threshold $T_{I\text{-}sh}$, it stays in Slow Run; or else, PE sleeps when the waited idle time $T_{I'}$ becomes larger than the threshold $T_{I\text{-}sh}$.

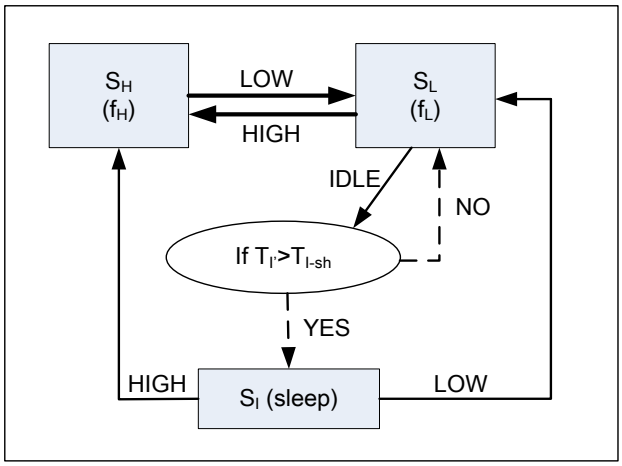


Figure 5-22 State transitions

The energy saving and delay for the adaption between the slow run state (S_L) and the sleep state (S_I) are presented with the waited idle time. If the waited idle time $T_{I'}<T_{I\text{-}sh}$ and buffer idle time $T_{BI}<T_{I\text{-}sh}$, the system stays in the S_L state, T_D=0, E_D=0. If the waited idle time $T_{I'}>T_{I\text{-}sh}$, then actual idle state time is $T_I = T_{BI} - T_{I\text{-}sh} - T_E$, and delay time is $T_D = T_{I\text{-}sh} + T_W$. But if $T_{I'} > T_{BI}$ and short idle time T_{BI} appears frequently, T_W would have an impact on system performance. The energy differences (E_D) is shown in Equation 5-3, which is the energy with low power P_{LR} minus the energy with idle power P_I and transition power P_T.

$$\begin{aligned} E_D &= P_{LR}\cdot(T_{BI}+T_W) - P_I\cdot T_I - P_T\cdot(T_E+T_W) \\ &= P_{LR}\cdot(T_{BI}+T_W) - P_I\cdot(T_{BI}-T_{I-sh}-T_E) - P_T\cdot(T_E+T_W) \\ &= (P_{LR}-P_I)\cdot T_{BI} + (P_{LR}-P_T)\cdot T_W + P_I\cdot T_{I-sh} - (P_T-P_I)\cdot T_E \end{aligned} \quad (5\text{-}3)$$

5.6 Power Aware Parallel System

This section presents the initial work on power aware parallel DPI system; the follow-up work and its implementation are the future work.

5.6.1 System Overview

The tremendous success of the Internet has made it a huge infrastructure that is connected with an enormous number of network devices to deliver information. Recently, low power Internet attracts more attention as people realize the significant amount power consumption for the underlying network infrastructure. In recent years, along with a continuous growth of both energy cost and network energy requirement, research and industry show big interests in energy efficient network technologies due to ecological and economical reasons.

DPI often cannot keep up with the fast increasing speed when it is deployed in core network. Compared to other algorithms (e.g. packet classification) for packet processing, DPI becomes a bottleneck at the router. In practice, a solution is to use multiple DPI systems in parallel, each of which handles part of traffic volume. The parallel system gains speed acceleration at the cost of duplicated memory and backup energy consumption. As the fast and stable speed is a fundamental goal of the initial Internet design, the cost of extra energy is worthwhile considering the peak time traffic. But energy can be saved during the low traffic period.

Parallel DPI can be designed with two operation modes, power-regardless mode and power-aware mode. Power-regardless design runs at the full capacity all the time, despite its fast processing speed and performance priority, it wastes a lot of energy. Considering the power-awareness, based on network characteristics, the system adapts between a single engine and multiple engines, by turning on or off particular engines. The functional switch of an engine depends on the detection of traffic condition and control policies.

For power efficiency, the power-aware design makes as many engines as possible into sleep state. The mechanism of system adaption is explained with the following example. Suppose the system supports four frequencies: f_1, f_2, f_3, f_4, where f_1 is the slowest, f_4 is the fastest and $f_1<f_2<f_3<f_4$; a “switch down” signal decreases to a lower frequency and a “switch up” signal increases to a higher frequency.

Depending on the pre-defined thresholds, an engine buffer is determined to be switched among the three statuses: High, Normal and Low. The priorities of *N* engines in a system are defined as follows:

Firstly, Engine 1 is considered as the main engine that always runs faster than f_1 and it should always run no slower than the other engines.

Secondly, the other engines, Engine 2 to Engine *N*, will wake up if their corresponding previous engines are working and the system cannot handle incoming traffic in the buffer even if all the active engines run at f_3 or f_4. Since when a new engine is brought in, all the other working engines must be busy, the next packet should be allocated to be processed by the new engine.

Thirdly, when the buffer of an engine stays at the Low stage for a period of time, it issues a "switch down" request signal to the scheduler.

Fourthly, when an engine stays in f_1 and the buffer of engine stays at the Low stage for some time, it issues a "sleep" request to the scheduler. When an engine is turned to sleep, the engines whose sequences are smaller than this engine are lifted up to make sure active engine sequences are larger than those of sleep engines.

One criterion of network application design is system stability. Based on general scale, two stable conditions are considered: idle mode for loose traffic and busy mode for peak traffic. In the idle mode, Engine 1 runs at f_2 or f_3 while the other engines sleep. In busy mode, each engine runs at f_2 or f_3, and Engine 1 should always be faster than the rest of engines.

In the busy mode, the issuing of a "switch down" signal targets at the goal of minimum overall power consumption. For example, when the buffer load is decreased, it first decreases the frequency of a slower engine, so that this engine can be turned to sleep later. On the other hand, when the buffer load is increased, the issuing of a "switch up" signal chooses the engine that can make the most effective performance improvement. For the purpose of reducing energy consumption, the traffic scheduler tries to turn more engines to the sleep state and do not wake them up unless the active

engines cannot handle current workload.

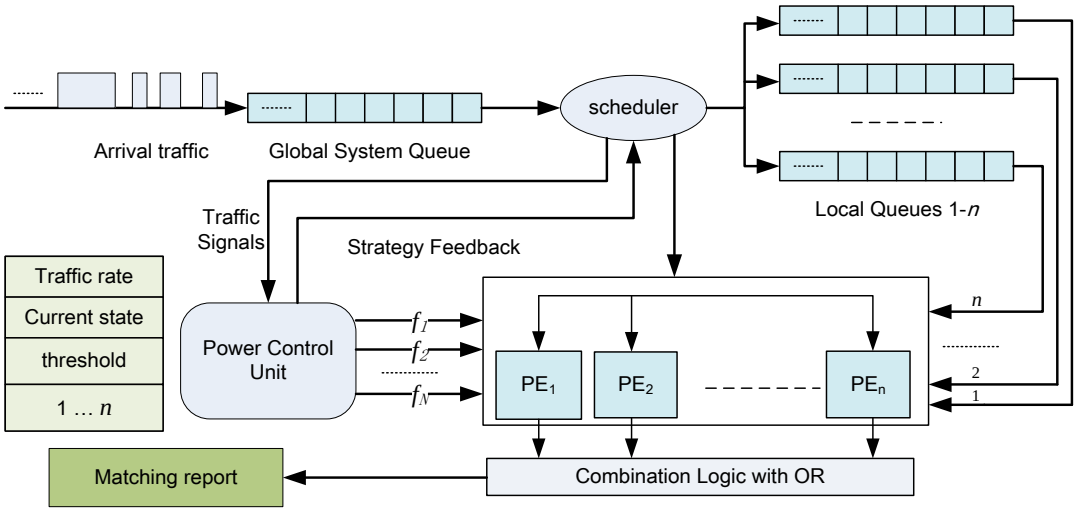


Figure 5-23 Parallel architecture with traffic scheduler and power control unit

Figure 5-23 presents an overview of the power-aware traffic scheduling method. There are three main components in power aware scheduling system: Traffic Scheduling Unit (TSU), Power Control Unit (PCU) and Processing Engine (PE). Deployed in high-speed core network routers, multiple PEs are needed when a single PE cannot handle the network traffic. Each PE runs a DPI engine for pattern matching on packet content; in addition, this system can also be extended to the low power design for a variety of network applications. In a parallel packet processing system, the multiple processing engines can be controlled independently. For example, some engines can sleep or different engines could run at different frequencies.

TSU is used for scheduling the high speed arriving traffic to multiple DPI engines in parallel. In particular, the traffic in the global queue is allocated to the local queues for each PEs. Instead of equally treating all the engines, the scheduling method also refers to current status of each engine for system power efficiency. A fundamental rule is that it will not wake up a sleeping engine unless all the working engines are already working in the high frequency. Basically, three schemes are considered including: 1) round robin for each engine; 2) performance priority scheduling; 3) power priority scheduling.

PCU is used for adapting the voltage and frequency supply of PEs to current workload. The voltage adaption controls a processing engine to work or sleep, and the frequency adaption controls a processing engine to work faster or slower. The status adaption for a processing engine depends on system traffic conditions indicated from FIFO status, and the interactive signals from TSU.

5.6.2 Scheduling

In high speed network, multiple parallel processing engines are used to provide high-performance processing during peak traffic period. Based on Figure 5-23, the objective of power-aware scheduling algorithm is the minimization of both the total energy consumption and the processing time of packet inspections on network traffic during a time period. Traditional approaches focus on a single objective; based on game theory, dual-objective optimization could be formulated as a cooperative game problem, where a bargaining point can be determined under different traffic conditions.

Power efficiency is achieved by maintaining a minimum number of engines at working state. For example, when there are multiple active engines, the new packet will not be allocated the engine with a low frequency, since the low frequency engine has larger chances to sleep in future. In addition, when system engines stays keep at one state for a period of time, it is considered as a stable situation, when the scheduling turns to be speed priority so long as does not change the effective energy consumption. The effective energy is the energy spent on working status, which does not include waiting or sleeping energy. In the case of speed priority, the packets are allocated to the engines with the smallest waiting and processing time.

The scheduling does not affect the total energy consumption of the system with fixed frequencies during a period of time. In Figure 5-24, the workload is allocated to engines buffers L_1 and L_2, associated with P_1 and P_2, respectively. Suppose that the power consumptions for running with full workload ($P_{1\text{-}run}$ and $P_{2\text{-}run}$) are m times as that of running with empty workload, which can be called “idle run” ($P_{1\text{-}idle}$ and $P_{2\text{-}idle}$).

Energy E during t time period is calculated in Equation 5-4, where ($P_1=P_{1\text{-}run}=m\cdot P_{1\text{-}idle}$), ($P_2=P_{2\text{-}run}=m\cdot P_{2\text{-}idle}$), and E is determined by the sum of workload ($L=L_1+L_2$), regardless of the allocation scheme.

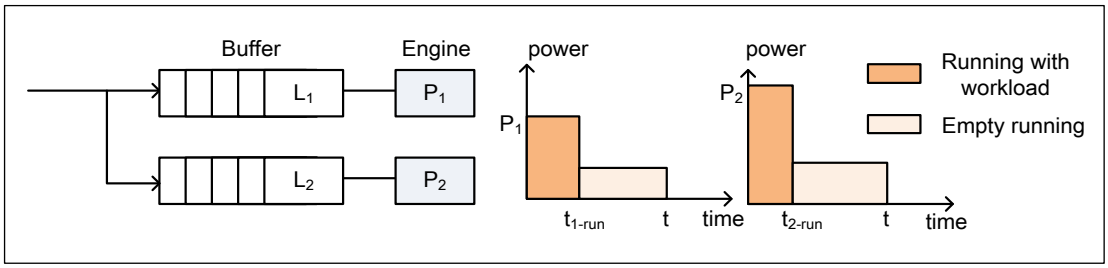


Figure 5-24 Energy consumption of engines with fixed frequencies

$$
\begin{aligned}
E &= t_{1\text{-}run}\cdot P_{1\text{-}run} + t_{1\text{-}idle}\cdot P_{1\text{-}idle} + t_{2\text{-}run}\cdot P_{2\text{-}run} + t_{2\text{-}idle}\cdot P_{2\text{-}idle} \\
&= L_1/P_1\cdot P_1 + (t-L_1/P_1)\cdot P_1/m + L_2/P_2\cdot P_2 + (t-L_2/P_2)\cdot P_2/m \qquad (5\text{-}4) \\
&= (L_1+L_2) + t\cdot(P_1+P_2)/m - (L_1+L_2)/m \\
&= L\cdot(1-1/m) + t\cdot(P_1+P_2)/m
\end{aligned}
$$

The scheduling for a system with dynamic frequencies will affect its energy and power consumption, which is considered based on three different objectives.

Objective 1: minimize time requirements given a power constraint.

The power constraint is assumed to be a factor of the maximum acceptable power. Given that it meets the power constraint, the processing time of packets will be minimized.

Objective 2: minimize the energy consumption with allowable processing time.

While throughput catches up with speed requirement, the schedule aims at the minimization of energy consumption.

Objective 3: an optimization of both processing time and energy constraints for the maximum benefits and the minimum overall penalty.

The scheduling is based on a joint objective; given a budget for energy and execution time plans, it would incur penalties if either or both requirements are not

satisfied.

1) Scheduling for Performance Priority

Problem Description:

Assume that there are *N* packets in the input buffer that will be processed by *M* available engines, and the packets have either the same or different processing times on a given engine. The aim is to assign packets to these engines so that the completion time is minimized.

Scenario 1: scheduling on identical or symmetric engines.

Given a set *J* of *N* packets with processing times t_i, i=1…N, and a positive integer *M*. The total processing time is $\sum_{i=1}^{N} t_i$, the minimum processing time is $min\left\{ \max_{1\le j\le M} \sum_{i=1}^{n_j} t_i \right\}$, where $\sum_{j=1}^{M} n_j = N$.

The scheme is scheduling with the next packets to the engine that has the least amount of local buffer load so far.

Scenario 2: scheduling on asymmetric engines.

Given a set *J* of *N* packets and a set of *M* engines, the processing times for packet $j \in J$ on engine $i \in M$, is t_{ij}. The total time *T* should meet the following requirements:

(1) One packet is completed on one engine. If packet *j* is processed on engine *i*, x_{ij} = 1, or else x_{ij} = 0. Therefore, for any $j \in J$, it meets $\sum_{i\in M} x_{ij} = 1$.

(2) T is the maximum processing time among all the engines. In other words, for any engine $i \in M$, $\sum_{j\in J} x_{ij} t_{ij} \le T$, where $x_{ij} \in \{0,1\}$. As a result, the minimum processing time is $min \max_{1\le j\le M} \sum_{i=1}^{N} t_{ij} x_{ij}$.

The scheme is scheduling with the fastest engine first, which can also be called as asymmetry-aware load balancing.

To make it simpler, suppose running at f_1 consumes half power as that of running

at f_2. Scheduling is basically the mapping of incoming packets to the buffer of a processing engine. The processing load depends on packet length, as an example, four load levels are considered, (L=1, 2, 3, 4). The processing speed is related with the running frequency at four processing levels, (P=1, 2, 3, 4). The required processing time at an engine is (t=L/P). The scheduling algorithm is based on certain frequencies at a given time.

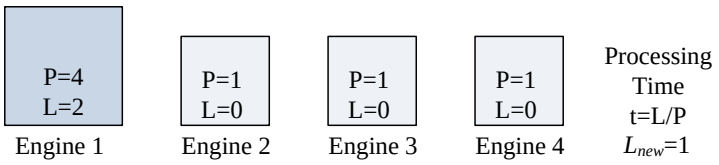


Figure 5-25 scheduling on asymmetric engines

As shown in Figure 5-25, if the next packet has load L_{new}=1: for scheduling with the faster core first, it will choose Engine 1; or else, it will choose Engine 2, 3 or 4. The processing durations required for a new load at the four engines are t_1=0.75, t_2=1, t_3=1, t_4=1, respectively. Engine 1 is a better choice whose load buffer is L=3. Running more packets on Engine 1 brings more benefits from the faster engine, leading to a higher throughput. Additionally, for another packet of L'_{new}=1, it should go to a slow engine (e.g. Engine 2, 3 or 4) since the faster is no longer faster with t'_1=1, which is the same as running on a slow engine.

2) Scheduling for Power Priority

This scheduling scheme works in cooperation with the power adaption scheme. Based on the above assumptions, suppose p_{ij} is the amount of power for packet i processed on engine j, the minimum total power consumption is described as $min\sum_{i=1}^{N}\sum_{j=1}^{M} p_{ij}x_{ij}$. The frequency f_j could vary from f_{min} to f_{max}, it affects the power and processing speed. The scheme is the energy aware packet scheduling.

3) Scheduling for a Tradeoff of Both Objectives

A bargaining point with $min \max_{1 \le j \le M} \sum_{i=1}^{n_j} t_i x_{ij}$ and $min \sum_{i=1}^{N} \sum_{j=1}^{M} p_{ij} x_{ij}$, the objective is to optimize the cumulative performance of all the engines rather than an individual engine.

Similarly, it should meet three constraints:

(a) $x_{ij} \in \{0,1\}$, whether packet *i* is mapped to engine *j*;

(b) $t_{ij} x_{ij} \le d_i$, d_i is the deadline of each packet processing;

(c) $\prod_{i=1}^{N} (t_{ij} x_{ij} \le d_i)$, j meets the condition that $x_{ij} = 1$, it constrains the deadline of all the packets are satisfied.

5.7 Summary

This chapter is the proposal for the future work of power modeling and detailed designs for low power DPI. In this chapter, a power saving model for routers with dynamic frequencies adapted to current traffic load is proposed using *M*/*D*/*s* queuing theory, where customers arrive in *Poisson* pattern and are served in constant time. Another model is proposed for DPI with dynamic frequencies using *M*/*M*/*s*, where customers arrive and are served in *Poisson* model. The difference is that router processing normally checks the packet header and DPI checks the whole packet payload. In future work, the models will be used to evaluate the contributions of power saving as well as the impact of waiting and processing time through dynamic frequency scaling.

Another problem that is usually neglected for dynamic frequency scaling is the fluctuation during the adaption with different frequencies, which is an overhead of power consumption and processing time. Through preliminary experiment with a router simulator for packet processing, the fluctuation can be alleviated by the use of two thresholds instead of a single threshold to control the adaption between two frequencies.

The low power DPI design is proposed based on the adaption between states with different frequencies and a sleep state. Power-aware scheduling among multiple engines is discussed for a parallel DPI system. Depending on current traffic load, a minimum number of engines are kept active while guaranteeing the required throughput. Currently, this chapter is limited to methods and designs; the evaluation and implementation of this chapter are my future work.

Chapter 6: Conclusions and Future Work

This chapter presents the conclusions drawn from this thesis and indicates two directions for future research work.

6.1 Summary

Many research efforts have been motivated by the tremendous growth of the Internet, aimed at improving the performance and security of web service. For network security as well as network management like traffic filtering, metering and monitoring, DPI successfully assists NIDS to safeguard the Internet. DPI can identify and classify traffic based on a pattern database by inspecting packet payload, and it provides a finer control than packet classification, which is based on packet header.

Although DPI has already been integrated in core network devices supported by a number of network equipment suppliers, the memory and processing complexities restrict its application. This thesis works on the memory compression, hardware acceleration and power efficiency of DPI algorithms for the network packet processing.

6.1.1 DFA Compression

DPI patterns can be represented with the fixed strings or the regular expressions. Regular expressions are more widely used due to their better expressive capability. DFA and NFA based algorithms are typically used to implement regular expression based pattern matching algorithms. In comparison, NFA is compact but has multiple active states; DFA has only one active state but may result in state explosion. DFA can be converted from NFA by associating with deterministic transitions and a unique state for each input character; however, some special constraints and combinations of substrings may lead to a substantial growth of the memory size.

In Chapter 3, the inter-state or intra-state redundancies in DFA transitions are analyzed for compression. After a comprehensive analysis of different redundancies in DFA structure, four types of redundancies are summarized with large compression potentials. Two improved DFA methods, Extend-D^2FA and Tag-DFA, are proposed to exploit more than one kind of redundancy to compress DFA transitions. Compared with a well-known algorithm D^2FA, experiments show that Extend-D^2FA has a larger compression ratio with the same number of default transitions as D^2FA. Furthermore, Tag-DFA achieves more than 90% of compression ratio and it requries no more than two states traversal for each character, rather than multiple default transitions in D^2FA. Comparing Extend-D^2FA and Tag-DFA, Extend-D^2FA can be easily built based on D^2FA while Tag-DFA needs to divide DFA states into groups and perform compression on each group. Despite a larger contruction complexity than Extend-D^2FA, Tag-DFA restricts the number of default transitions for each input character.

6.1.2 Bloom Filter

Up to now, the software-based pattern matching algorithms cannot reach the high-speed throughput at core routers; therefore, many hardware-based algorithms have been proposed, most of which are implemented on FPGA. The states and transitions information in automaton can be stored in both on-chip and off-chip memory. The hardware acceleration is mostly achieved by exploring the parallel implementation of DPI algorithms with independent processing engine.

As a computing intensive component, Bloom filters are widely used for network packet processing, and are suitable for hardware implementation in parallel. Bloom filters work fast with reasonable memory requirements. One difficulty for its application in DPI is that Bloom filters are not suitable for the matching of long patterns, and a more memory efficient solution is needed, such as using fingerprints rather than storing the whole patterns for comparison.

In Chapter 4, the hardware implementation of the multi-pattern matching using

parallel engines of CBF is presented, which is a space and time efficient data structure, and it allows optimal match with counters by comparing with associated patterns. To reduce memory requirement, pruning and list-balancing techniques are applied to CBF.

Secondly, an energy efficient adaptive Bloom filter, EABF, is devoted to a balance of power and performance especially for high performance networks. The basic idea is to give Bloom Filters the capability of adjusting the number of active hash functions according to current workload automatically. The adaptive procedure depends on its control policies and in this study, three policies are presented and compared. For high performance implementation of EABF in hardware, a method is also presented in a two-stage platform, where Stage 1 is always active and Stage 2 sleeps until Stage 1 reports a positive matching. The platform is flexible and can be extended to multi-stage or the full pipelined *k*-stage. A control circuit is designed for flexibly changing working stage and reducing both dynamic and static power consumptions. Analysis and experiments show that our EABF have almost the best power savings compared to the fixed schemes; on the other hand, EABF achieves a performance close to the optimal one clock cycle latency on average, compared to a much longer latency of fixed schemes.

Bloom filter is widely used in network packet processing due to its fast lookup speed and small memory cost. However, the non-negligible false positive rate and the difficulty of online update still prevent it from extensive utilization. A cache-based counting Bloom filter architecture, C^2BF, is proposed, which is not only easy to update online but also benefical for fast verification for precise matching. Besides, a high speed hardware C^2BF architecture with off-chip memory and fast cache replacement method is presented. C^2BF has three contributions: 1) compressed CBF implementation and its updating algorithm; 2) pattern grouping for higher cache hit rate; 3) on-chip cache organization and replacement policy. Experiments show that our prototype of C^2BF reduces more than 70% of the verification processing time with cache design compared to traditional non-cache schemes without cache.

6.2 Future Work

The objective of this research focuses on energy aware pattern matching algorithms with hardware acceleration. My future work includes the follow-up work of optimization of Tag-DFA generation in Section 3.5, the hardware implementation of EABF on FPGA, and the evaluation of the models and low power design methods in Chapter 5. In addition, the author has two other goals in future work: mobile Internet and power aware network protocol.

Mobile Internet is a fast growing business with great market potential especially with the widespread of smart phones. However, the exploding of data and video traffic affects service quality, charging policy and brings security threats. Mobile DPI is now becoming more and more necessary for mobile carriers. It can be used for fine-grained application detection, so that carriers can monitor and control traffic flows, optimize bandwidth and make specified charging policies. Different from traditional DPI, the main aim of mobile DPI is the application detection. The signatures are used to identify particular behaviours of applications.

The energy efficient algorithms will be extended to multiple routers and to the protocol level. At the network level, the low power algorithm can also take advantage of the cooperation among multiple routers. A previous router can briefly notify the size of incoming traffic flows to other routers on the same link, so that the subsequent routers can adjust their working frequencies.

6.2.1 Power-aware DPI Design

Chapter 5 proposes the power models and the designs of low power DPI, the evaluation of which is my future work. At the router and DPI levels, in order to estimate the contribution of power savings, power saving model will be built using queuing theory to describe the dynamic frequency scaling suited to the current traffic load. In addition, another model will be built to illustrate the solution to the frequency fluctuation in dynamic frequency scaling. At the system level, low power DPI will be

implemented using the low power designs in Section 5.5 and Section 5.6, which have multiple states for modes with different frequencies as well as a sleep mode. The power-aware scheduling among multiple engines is considered in a parallel DPI system. Depending on arrival traffic load, a minimum number of engines will be kept active while guaranteeing the required throughput.

6.2.2 Mobile DPI for Fine-grained Application Detection

- **Mobile Internet Market**

Mobile communications and Internet are regarded as two of the world's fastest growing businesses. Especially nowadays, with a wide range of mobile devices, e.g. smart phones and iPads, people could encounter a large variety of emerging mobile Internet applications at all times and places. Statistics show that over 1 billion of the worlds 4+ billion mobiles phones are now smart phones (over 27%), and 3 billion are SMS enabled (75%). In 2014, mobile internet usage will overtake desktop internet usage; and already in 2011, more than 50% of all “local” searches are done from a mobile device. In particular, 86% of mobile users are watching TV while using a mobile phone, 200+ million (1/3 of all users) access Facebook from a mobile device and 91% of all mobile internet use is “social” related.

Customers spend a lot of time on the mobile Internet. On average, Americans spend 2.7 hours per day socializing on their mobile devices. That is over twice the amount of time they spend on eating, and over one third of the time they spend on sleeping each day. Besides, 91% of mobile Internet access is to socialize, which is compared to 79% of users on desktops.

The top three popular social networking sites, Twitter has almost doubled its Irish accounts, Facebook is adding 900 new Irish profiles per day. LinkedIn is growing at a much faster than Facebook, in percentage terms at least 11% versus 4%. With the fast development mobile Internet, people prefer to log in through Smartphone and update their information anywhere anytime. Over one third of

Facebook's over 600 million user base uses Facebook mobile, half of Twitter's 165 million users use Twitter mobile. Over 200 million users view Youtube on mobile devices per day. On average, 30% of smartphone owners accessed social networks via mobile browser.

Mobile Internet becomes a very attractive business with extremely fast growing speed and great market potential all over the world. According to a survey [100], China's mobile Internet market reached 7.79 billion RMB in the second quarter of 2011. Various new applications sprung up in the past several years to satisfy user demand, some of them are very popular while others might be a waste of resources. The explosion of mobile traffic also brings security threats which might leak private information on phones. Thus further analysis of applications in the traffic is very helpful for both service providers and terminal users.

However, there hasn't been an effective system for application detection in mobile network. Traditional fixed line Internet uses DPI for accurate traffic analysis, which is usually integrated in Network Intrusion Detection Systems (NIDS). Some mature tools, e.g. OpenDPI [101], can identify specific applications such as MSN, Opera and Facebook. Accordingly, mobile DPI is motivated to analyze application behavior in the mobile traffic. The mobile DPI modules can be deployed in wireless controllers, base stations or network cores. On the one hand, the mobile DPI aims at secure and better service for users. On the other hand, it works for bandwidth optimization and more sensible charging schemes for a larger profit of service providers. For example, different charging schemes can be employed by investigating which broadband applications are most popular with which level of subscriptions.

- **Framework of Application Detection**

The main goal of mobile DPI system is to detect and collect fine-grained application behavior in mobile Internet. As shown in Figure 6-1, the fine-grained application detection framework has four main modules, including the traffic handling module, the rule management module, the application detection module and the demo

module.

Firstly, the traffic handling module captures packets from online traffic and processes packet header with connection table. The connection table contains all the information about previous connection flows. Second, the rule management module initializes and maintains the two state machines for application detection. The rules formulate application type, domains etc. aspects and are represented by regular expression or behavior sequence. Third, the application detection module is a key component to process packet payload. This module uses regular expressions to inspect individual payload data, and uses behavior sequence detection to check continuous packet payloads against application usage patterns. DFA based state machines are used for time efficiency. Fourth, the detection results are presented by the demo module for three scenarios: 1) application and flow monitoring for system management and bandwidth optimization; 2) user behavior statistics for personalized services; 3) malware detection and filtering for safeguarding mobile Internet.

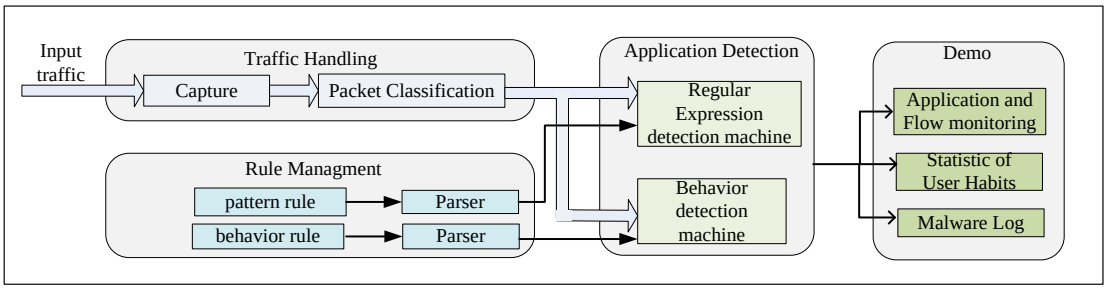


Figure 6-1 Framework of Application Detection in Mobile DPI

Basically, the system works as follows. The system initializes with user defined rules to capture online or offline traffic. The application detection module processes each incoming packet and updates the connection table. Two matching state machines parse packet payload through user-defined rules. The system generates security alerts if any malicious traffic is detected and makes statistic report correspondingly.

Future work will be performed to support more applications. As this work is still in exploration stage and the matching signatures are not mature, the detection accuracy is crucial to mobile DPI. Effective methods for application detection will be investigated. Besides, similar as traditional DPI, memory consumption and matching

speed are also important.

6.2.3 Other Power-aware Research on Green Internet

- **Support to smart standby mode**

As stated in [95][96], the feasible approaches to reduce power consumption of a router can be classified as dynamic power scaling and smart standby. Sleeping and standby approaches allow devices or parts of them turning themselves almost completely off, and entering very low energy states, when all their functionalities are frozen [6]. Network devices cannot completely sleep since they need to maintain the "network presence"; otherwise they will become unreachable and fall off the network. Normal protocols require switches and router to respond even when it is not at work.

Therefore, the real deployment of sleeping needs the support of network protocols or needs to be assisted by proxies. For example, as an alternative to the necessity of "network presence", Christensen et al. in [102] and [103], explore the idea of using a proxy to "cover" a network host. When the network host sleeps during empty traffic load, the proxy responds to ARP packets in order to maintain the reach ability from the router, and to respond to other protocol and application messages, which are needed to maintain the full network presence.

IEEE Energy Efficient Ethernet Task Force has also started to explore both sleeping and rate adaptation for energy savings. It can be expected that the sleeping mode of a router will be included in future network standard [104].

- **Network Infrastructure or Routing Protocol**

In summary, a fundamental method to reduce network energy consumption is by allowing some links, entire network devices, or parts thereof to a low power mode or a sleep mode in a smart and effective way during light traffic period.

The algorithms in this thesis mainly concerns about the power efficiency of a single router; at the network level, multiple routers can cooperate together by sharing

their traffic flow information. Besides forwarding network packets, a brief report of upcoming traffic can be sent in advance to the next hop router to assist its power control. For example, if there will be abundant packets coming soon, a router receives an alert from a previous router on the link, and pre-adjusts its working state.

Furthermore, application analysis on packet header or packet payload is another way to predict future network flows. The major type of applications include video stream, P2P file sharing and web data etc. Different kinds of applications have different traffic distribution characteristics.

The fundamental transform of next generation Internet to energy efficiency must be considered from a larger perspective [105][106][107][108]. For example, Green reconfigurable router proposed by Zhang et al. in [109] shows low power design from traffic engineering at the network level. Our future work will focus on the power awareness at the network level, and study the power saving opportunities in the network infrastructure and the network protocols.

The router is designed with multiple stages, each of which has a different frequency. The switching of working stage depends on trace size; however, the traffic monitoring function itself in a router consumes extra energy. Another approach is to use some particular routers for trace detection and sends notifications to other routers, when a large trace is forwarding to these routers. The trace size information will assist router stage adaption. Current network infrastructure exhibits poor power efficiency, running all the time at full capacity, regardless of the traffic demand and distribution over the Internet. Considering multiple routers in the network infrastructure, the pre-caution mechanism would further facilitate the power savings for core routers and edge routers. Basically, a notification can be sent to the edge router before the arrival of a large trace. This function can be integrated in the new routing protocol.

The power-awareness can be also included in the establishment of links between source and destination terminals, so that more routers can run at lower frequencies or support smart standby. Moreover, the smart standby of routers also requires the support of certain network protocols. The improvement of routing protocols is to

facilitate power management at the network level by routing traffic through different paths to adjust the workload on individual routers or links.

References

[1] "Internet World Stats - Usage and Population Statistics." [Online]. Available: http://www.internetworldstats.com/. [Accessed: 28-Mar-2012].

[2] "Visual Networking Index - Cisco Systems." [Online]. Available: http://www.cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html. [Accessed: 28-Mar-2012].

[3] "Internet Bandwidth Growth: Dealing with Reality." [Online]. Available: http://www.isoc.org/isoc/conferences/bwpanel/. [Accessed: 28-Mar-2012].

[4] "Infographic: Mobile Statistics, Stats & Facts 2011." [Online]. Available: http://www.digitalbuzzblog.com/2011-mobile-statistics-stats-facts-marketing-infographic/. [Accessed: 28-Mar-2012].

[5] R. Bolla, R. Bruschi, F. Davoli, and F. Cucchietti, "Energy efficiency in the future internet: a survey of existing approaches and trends in energy-aware fixed network infrastructures," *Communications Surveys & Tutorials, IEEE*, no. 99, pp. 1–22, 2010.

[6] K. Christensen, F. Cucchietti, and S. Singh, "The Potential Impact of Green Technologies in Next-Generation Wireline Networks: Is There Room for Energy Saving Optimization?," *IEEE Communications Magazine*, p. 81, 2011.

[7] Symantec Company, "Internet Security Threat Report, April 2012." [Online]. Available: http://www.symantec.com/threatreport/. [Accessed: 24-May-2012].

[8] Gartner, "Worldwide Sales of Mobile Phones in First Quarter of 2012." [Online]. Available: http://www.gartner.com/it/page.jsp?id=2017015. [Accessed: 24-May-2012].

[9] "SNORT Network Intrusion Detection System." [Online]. Available: http://www.snort.org/. [Accessed: 28-Mar-2012].

[10] "Bro Intrusion Detection System - Bro Overview." [Online]. Available: http://www-old.bro-ids.org/. [Accessed: 28-Mar-2012].

[11] "L7-filter." [Online]. Available: http://l7-filter.sourceforge.net/. [Accessed: 28-Mar-2012].

[12] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2004, vol. 1, pp. 485–509.

[13] "Clam AntiVirus signature database." [Online]. Available: http://www.clamav.net/lang/en/. [Accessed: 28-Mar-2012].

[14] M. Necker, D. Contis, and D. Schimmel, "TCP-stream reassembly and state tracking in hardware," in *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, 2002, pp. 286–287.

[15] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Macia-Fernandez, and E. Vazquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1–2, pp. 18–28, 2009.

[16] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 339–350, 2006.

[17] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *ACM/IEEE Symposium on Architecture*

for Networking and Communications systems, ANCS 2006., 2006, pp. 93–102.

[18] D. Guo, G. Liao, L. N. Bhuyan, and B. Liu, "An adaptive hash-based multilayer scheduler for L7-filter on a highly threaded hierarchical multi-core server," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2009, pp. 50–59.

[19] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, 2006, pp. 81–92.

[20] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved DFA for fast regular expression matching," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.

[21] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the 2007 ACM CoNEXT conference*, 2007, pp. 1–12.

[22] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *IEEE Symposium on Security and Privacy, SP 2008.*, 2008, pp. 187–201.

[23] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[24] D. E. Knuth, J. H. Morris Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[25] Z. Baker and V. Prasanna, "Automatic synthesis of efficient intrusion detection systems on FPGAs," *Field Programmable Logic and Application*, pp. 311–321, 2004.

[26] R. N. Horspool, "Practical fast searching in strings," *Software: Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[27] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[28] G. A. Stephen, *String searching algorithms*, vol. 3. World Scientific Pub Co Inc, 1994.

[29] M. Ramakrishna, E. Fu, and E. Bahcekapili, "A performance study of hashing functions for hardware applications," *Proc. ICCI'94*, vol. 5, pp. 1621–1636, 1994.

[30] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*. University of California Press, 1997.

[31] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, 2004, vol. 4, pp. 2628–2639.

[32] X. Ke, C. Yong, and others, "An improved Wu-Manber multiple patterns matching algorithm," in *25th IEEE International Performance, Computing, and Communications Conference. IPCCC 2006.*, 2006, p. 6–pp.

[33] L. Tan, B. Brotherton, and T. Sherwood, "Bit-split string-matching engines for intrusion detection and prevention," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 3, no. 1, pp. 3–34, 2006.

[34] N. Sertac Artan and H. J. Chao, "TriBiCa: Trie bitmap content analyzer for high-speed network intrusion detection," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, 2007, pp. 125–133.

[35] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet

inspection using parallel bloom filters," in *Proceedings of 11th Symposium on High Performance Interconnects,*, 2003, pp. 44–51.

[36] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'01.*, 2001, pp. 227–238.

[37] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, 2007, pp. 155–164.

[38] N. Hua, H. Song, and T. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," in *INFOCOM 2009, IEEE*, 2009, pp. 415–423.

[39] L. Vespa, N. Weng, and R. Ramaswamy, "MS-DFA: Multiple-Stride Pattern Matching for Scalable Deep Packet Inspection," *The Computer Journal*, vol. 54, no. 2, pp. 285–303, 2011.

[40] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, "A memory-efficient parallel string matching architecture for high-speed intrusion detection," *IEEE Journal on Selected Areas in Communications,*, vol. 24, no. 10, pp. 1793–1804, 2006.

[41] Y. Sugawara, M. Inaba, and K. Hiraki, "Over 10gbps string matching mechanism for multi-stream packet scanning systems," *Field Programmable Logic and Application*, pp. 484–493, 2004.

[42] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ACM SIGARCH Computer Architecture News*, 2006, vol. 34, pp. 191–202.

[43] "Regular Expression Processor - Regex_wiki." [Online]. Available: http://regex.wustl.edu/index.php/Main_Page. [Accessed: 28-Mar-2012].

[44] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, 2007, pp. 145–154.

[45] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, 2008, p. 1.

[46] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proceedings of the 2008 ACM CoNEXT Conference*, 2008, p. 25.

[47] G. Cormode and M. Thottan, *Algorithms for next generation networks*. Springer-Verlag New York Inc, 2010.

[48] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *ACM SIGCOMM Computer Communication Review*, 2008, vol. 38, pp. 207–218.

[49] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008, pp. 166–170.

[50] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[51] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web

cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[52] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer, "Single-packet IP traceback," *IEEE/ACM Transactions on Networking (ToN)*, vol. 10, no. 6, pp. 721–734, 2002.

[53] K. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-code bloom filter for efficient per-flow traffic measurement," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, 2004, vol. 3, pp. 1762–1773.

[54] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," *Algorithms–ESA 2006*, pp. 684–695, 2006.

[55] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Multilayer compressed counting bloom filters," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008, pp. 311–315.

[56] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, "Dynamic count filters," *ACM SIGMOD Record*, vol. 35, no. 1, pp. 26–32, 2006.

[57] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Transactions on Networking (TON)*, vol. 10, no. 5, pp. 604–612, 2002.

[58] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 241–252.

[59] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic bloom filters," in *Proc. 25th IEEE INFOCOM*, 2006, vol. 1, pp. 1–12.

[60] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic Bloom filters," *IEEE Transactions on Knowledge and Data Engineering,*, vol. 22, no. 1, pp. 120–133, 2010.

[61] S. Kumar and P. Crowley, "Segmented hash: an efficient hash table implementation for high performance networking subsystems," in *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, 2005, pp. 91–103.

[62] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008, pp. 101–105.

[63] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, 2004, pp. 174–183.

[64] D. Guo, G. Liao, L. N. Bhuyan, B. Liu, and J. J. Ding, "A scalable multithreaded L7-filter design for multi-core servers," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 60–68.

[65] "Power Compiler, Automatic Power Management within Galaxy." [Online]. Available: http://www.synopsys.com/tools/implementation/rtlsynthesis/pages/powercompiler.aspx. [Accessed: 28-Mar-2012].

[66] W. Liao, J. M. Basile, and L. He, "Microarchitecture-level leakage reduction with data retention," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,*, vol. 13, no. 11, pp. 1324–1328, 2005.

[67] M. A. Franklin and T. Wolf, "Power considerations in network processor design," *Network Processor Design: Issues and Practices*, vol. 2, pp. 29–50, 2003.

[68] "Cacti The Complete RRDTool-based Graphing Solution." [Online]. Available:

http://www.cacti.net/. [Accessed: 28-Mar-2012].

[69] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” in *ACM SIGARCH Computer Architecture News*, 2000, vol. 28, pp. 83–94.

[70] “LabJack for power measurement.” [Online]. Available: http://labjack.com/products. [Accessed: 26-Apr-2012].

[71] “PowerPlay Early Power Estimators (EPE) and Power Analyzer.” [Online]. Available: http://www.altera.com/support/devices/estimator/pow-powerplay.jsp. [Accessed: 15-May-2012].

[72] “Performing Power Analysis with the PowerPlay Power Analyzer.” [Online]. Available: http://quartushelp.altera.com/11.1/mergedProjects/optimize/pwr/pwr_pro_power_analyzer.htm. [Accessed: 28-Mar-2012].

[73] “Spartan-6 FPGA Power Management User Guide.” [Online]. Available: http://www.xilinx.com/products/technology/power/index.htm. [Accessed: 28-Mar-2012].

[74] L. Benini, A. Bogliolo, and G. De Micheli, “A survey of design techniques for system-level dynamic power management,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,*, vol. 8, no. 3, pp. 299–316, 2000.

[75] Y. Luo, J. Yu, J. Yang, and L. N. Bhuyan, “Conserving network processor power consumption by exploiting traffic variability,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 1, p. 4, 2007.

[76] H. Li, S. Bhunia, Y. Chen, T. Vijaykumar, and K. Roy, “Deterministic clock gating for microprocessor power reduction,” in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, 2003, pp. 113–122.

[77] “Stirling numbers of the second kind.” [Online]. Available: http://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind. [Accessed: 31-Mar-2012].

[78] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, “Fast hash table lookup using extended bloom filter: an aid to network processing,” in *ACM SIGCOMM Computer Communication Review*, 2005, vol. 35, pp. 181–192.

[79] S. Dharmapurikar and J. W. Lockwood, “Fast and scalable pattern matching for network intrusion detection systems,” *IEEE Journal on Selected Areas in Communications,*, vol. 24, no. 10, pp. 1781–1792, 2006.

[80] J. Baliga, K. Hinton, and R. S. Tucker, “Energy consumption of the Internet,” in *Joint International Conference on Optical Internet, 2007 and the 2007 32nd Australian Conference on Optical Fibre Technology. COIN-ACOFT 2007.*, 2007, pp. 1–3.

[81] “FP7 ECONET Project.” [Online]. Available: https://www.econet-project.eu/. [Accessed: 28-Mar-2012].

[82] R. Bolla, R. Bruschi, F. Davoli, and F. Cucchietti, “Energy efficiency in the future internet: a survey of existing approaches and trends in energy-aware fixed network infrastructures,” *Communications Surveys & Tutorials, IEEE*, no. 99, pp. 1–22, 2010.

[83] T. Kocak and I. Kaya, “Low-power bloom filter architecture for deep packet inspection,” *Communications Letters, IEEE*, vol. 10, no. 3, pp. 210–212, 2006.

[84] M. Paynter and T. Kocak, “Fully pipelined bloom filter architecture,” *Communications Letters, IEEE*, vol. 12, no. 11, pp. 855–857, 2008.

[85] E. Safi, A. Moshovos, and A. Veneris, “L-CBF: a low-power, fast counting Bloom filter architecture,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,*, vol. 16, no. 6, pp. 628–638, 2008.

[86] H. Song and J. W. Lockwood, “Multi-pattern signature matching for hardware network intrusion detection systems,” in *IEEE Global Telecommunications Conference, GLOBECOM’05.*, 2005, vol. 3, p. 5–pp.

[87] I. Kaya and T. Kocak, “A low power lookup technique for multi-hashing network applications,” in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, 2006, pp. 179–184.

[88] “IEEE 802.3 Energy Efficient Ethernet Study Group.” [Online]. Available: http://www.ieee802.org/3/eee_study/index.html. [Accessed: 28-Mar-2012].

[89] R. Merritt, “Energy-efficient Ethernet standard gains traction.” [Online]. Available: http://www.eetimes.com/electronics-news/4076984/Energy-efficient-Ethernet-standard-gains-traction. [Accessed: 28-Mar-2012].

[90] HP, “Energy Efficient Networking - Business whitepaper,” 2011.

[91] B. Zhang, J. Yang, and J. Wu, “Survey and Analysis on Internet Traffic Model,” *Journal of Software*, vol. 22, no. 1, 2011.

[92] H. C. Tijms and J. Wiley, *A first course in stochastic models*, vol. 2. Wiley Online Library, 2003.

[93] “Packet Length Distributions.” [Online]. Available: http://www.caida.org/research/traffic-analysis/AIX/plen_hist/. [Accessed: 28-Mar-2012].

[94] “M/M/1 Queueing System.” [Online]. Available: http://www.eventhelix.com/realtimemantra/congestioncontrol/m_m_1_queue.htm. [Accessed: 28-Mar-2012].

[95] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall, “Reducing network energy consumption via sleeping and rate-adaptation,” in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, pp. 323–336.

[96] M. Gupta and S. Singh, “Greening of the Internet,” in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 19–26.

[97] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, “Opportunities and challenges for better than worst-case design,” in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, 2005, pp. 2–7.

[98] S. P. Mohanty and N. Ranganathan, “Energy-efficient datapath scheduling using multiple voltages and dynamic clocking,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 10, no. 2, pp. 330–353, 2005.

[99] A. Kennedy, X. Wang, Z. Liu, and B. Liu, “Low power architecture for high speed packet classification,” in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 131–140.

[100] “Research and Markets: China Cell Phone Market Profile 2011 | Reuters.” [Online]. Available: http://www.reuters.com/article/2011/03/02/idUS27070+02-Mar-2011+BW20110302. [Accessed: 28-Mar-2012].

[101] “OpenDPI.org.” [Online]. Available: http://www.opendpi.org/. [Accessed: 28-Mar-2012].

[102] M. Jimeno, K. Christensen, and B. Nordman, “A network connection proxy to enable hosts to sleep and save energy,” in *IEEE International Performance, Computing and Communications Conference, IPCCC 2008.*, 2008, pp. 101–110.

[103] K. J. Christensen and F. B. Gulledge, “Enabling power management for network-attached computers,” *International Journal of Network Management*, vol. 8, no. 2, pp. 120–130, 1998.

[104] C. Gunaratne, K. Christensen, B. Nordman, and S. Suen, “Reducing the energy consumption of Ethernet with adaptive link rate (ALR),” *IEEE Transactions on Computers,*, vol. 57, no. 4, pp. 448–461, 2008.

[105] M. Gupta and S. Singh, “Using low-power modes for energy conservation in Ethernet LANs,” in *IEEE INFOCOM*, 2007, pp. 2451–2455.

[106] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright, “Power awareness in network design and routing,” in *The 27th Conference on Computer Communications IEEE INFOCOM 2008.*, 2008, pp. 457–465.

[107] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, “ElasticTree: Saving energy in data center networks,” in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010, p. 17–17.

[108] C. Hu, C. Wu, W. Xiong, B. Wang, J. Wu, and M. Jiang, “On the design of green reconfigurable router toward energy efficient internet,” *Communications Magazine, IEEE*, vol. 49, no. 6, pp. 83–87, 2011.

[109] M. Zhang, C. Yi, B. Liu, and B. Zhang, “GreenTE: Power-aware traffic engineering,” in *18th IEEE International Conference on Network Protocols (ICNP)*, 2010, pp. 21–30.

[110] iMesh application, [Online].
Available: http://protocolinfo.org/wiki/IMesh. [Accessed: 16-July-2012].