Energy Efficient Hardware Accelerators for Packet Classification and String Matching

by

Alan Kennedy, B.Eng.

Submitted in partial fulfilment of the requirements
for the Degree of Doctor of Philosophy

Dublin City University
School of Electronic Engineering
Supervisor: Dr. Xiaojun Wang
September 2010

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, 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:____________________

Alan Kennedy (Candidate)

ID No.:____________________

Date: ____________________

ACKNOWLEDGMENTS

Firstly, I would like to thank my supervisor Dr. Xiaojun Wang for taking me on as his student and giving me a great deal of help, support and guidance during my time in Dublin City University. I would also like to thank my colleagues in the Network Processing Group for making the lab an enjoyable place to work and for giving me their technical advice whenever needed. My gratitude also goes to Prof. Bin Liu and the other members of the Broadband Switching Laboratory in Tsinghua University with whom I collaborated.

I would like to give a special thanks to my parents for making this work possible by giving me their love, help and financial support during my seemingly never-ending education. Finally I would like to thank Lisa for her love, help and encouragement during the duration of my Ph.D., and for her understanding of the long hours spent in the lab.

TABLE OF CONTENTS

Declaration ..... **i**

Acknowledgments ..... **ii**

Table of Contents ..... **iii**

Abstract ..... **vii**

List of Figures ..... **viii**

List of Tables ..... **xi**

List of Acronyms ..... **xii**

Publications and Patents Arising from Work ..... **xiv**

Chapter 1 - Introduction **1**

1.1 Motivation 1

1.2 Network Overview 3

1.3 Packet Processing Bottlenecks 6

1.3.1 Packet Classification 7

1.3.2 Deep Packet Inspection 8

1.3.3 Technical Challenges 9

1.4 Contributions 11

1.5 Thesis Organisation 13

1.6 Summary 14

Chapter 2 - Background **15**

2.1 Introduction 15

2.2 Packet Classification Rulesets 16

2.3 Analysis of Software Approaches to Packet Classification 17

2.3.1 Algorithmic Approaches 18

2.3.2 Simulation Framework 24

2.3.3 Performance Results ... 25
2.3.4 Conclusions ... 30
2.4 Deep Packet Inspection Systems ... 30
2.4.1 Snort ... 31
2.4.2 Current Fixed String Matching Approaches ... 32
2.4.3 Conclusions ... 34
2.5 Hardware-Based Platforms ... 34
2.5.1 ASIC ... 35
2.5.2 FPGA ... 35
2.5.3 TCAM ... 36
2.5.4 Conclusions ... 38
2.6 Low Power Design ... 38
2.6.1 Types of Power Dissipation ... 39
2.6.2 Power Benchmarking ... 43
2.6.3 Low Power Design Techniques ... 44
2.7 Summary ... 47

Chapter 3 - Packet Classification Architectures ... 48

3.1 Introduction ... 48
3.2 Decision Tree-Based Packet Classification ... 49
3.2.1 Building a Decision Tree ... 51
3.2.2 Heuristics Used to Reduce Memory Usage ... 54
3.3 Algorithmic Modifications ... 56
3.3.1 Cutting Scheme ... 57
3.3.2 Region Compaction ... 59
3.3.3 Rule Storage ... 62
3.4 Cut Selection ... 64
3.5 Memory Organisation ... 66
3.5.1 Ultra-Wide Memory Words ... 66
3.5.2 Reduced Width Memory Words ... 68
3.6 Packet Classification Engine ... 73
3.6.1 Architecture of Engine Using Ultra-Wide Memory Words ... 74
3.6.2 Architecture of Engines Using Reduced Width Memory Words ... 77
3.7 Configuration of Multiple Engines Operating in Parallel ... 82
3.7.1 Architecture of Packet Buffer ... 83
3.7.2 Architecture of Sorter Logic Block ... 84
3.7.3 Architecture of Classifier Using Ultra-Wide Memory Words ... 85

3.7.4 Architecture of Classifier Using Reduced Width Memory Words 87
3.8 Performance Results 89
3.8.1 Hardware Implementation Parameters 89
3.8.2 Memory Usage and Worst Case Number of Memory Accesses 91
3.8.3 Throughput vs. Power Consumption 94
3.8.4 Evaluation Against Prior Art 97
3.9 Summary of Contributions 100

Chapter 4 - Frequency Scaling Architecture 102
4.1 Introduction 102
4.2 Analysis of Real Traces 103
4.2.1 Processing Needs 105
4.2.2 Classifier Utilisation 106
4.3 Methods for Reducing Power Consumption 107
4.3.1 Clock Gating/Turning Off Unused Processing Elements 107
4.3.2 Voltage/Frequency Scaling 108
4.4 Adaptive Clocking Scheme 109
4.4.1 Method for Reducing Frequency Switching 110
4.4.2 Adaptive Clocking Unit Architecture 113
4.5 Low Power Architecture for Packet Classification 115
4.5.1 Hardware Implementation Parameters 115
4.5.2 Power Consumption 117
4.6 Performance Testing Using Synthetic Traces 122
4.6.1 Power Savings 124
4.7 Summary of Contributions 126

Chapter 5 - String Matching Architecture 128
5.1 Introduction 128
5.2 String Matching Using Deterministic Finite Automaton 129
5.3 Memory Reduction 131
5.3.1 DFA Memory Usage Observations 132
5.3.2 Insertion of Default Transition Pointers 133
5.3.3 Algorithm for Building Search Structure 138
5.4 Memory Organisation and Hardware Architecture 142
5.4.1 Memory Layout 142
5.4.2 Hardware Accelerator Architecture 144
5.4.3 String Matching Engine Architecture 147

5.4.4 String Matching Scheduler Architecture 150
5.5 Performance Results 151
5.5.1 Characteristics of Snort Ruleset Used in Testing 151
5.5.2 Hardware Implementation Parameters 152
5.5.3 Transition Pointer Reduction 154
5.5.4 Throughput vs. Power Consumption 157
5.5.5 Evaluation Against Prior Art 158
5.6 Summary of Contributions 160

Chapter 6 – Conclusions and Future Work 161
6.1 Conclusions 161
6.1.1 Motivation for Proposed Research – A Summary 161
6.1.2 Summary of Thesis Contributions 162
6.1.3 Packet Classification 162
6.1.4 Frequency Scaling 164
6.1.5 String Matching 164
6.2 Future Work 165
6.2.1 Multi-Match Packet Classification 166
6.2.2 Regular Expression Matching 166
6.2.3 Reducing the Fixed String Matching Hardware Accelerator's Power .. 167

Appendix A – Power Usage 168

Bibliography 170

ABSTRACT

Energy Efficient Hardware Accelerators for Packet Classification and String Matching

Alan Kennedy

This thesis focuses on the design of new algorithms and energy efficient high throughput hardware accelerators that implement packet classification and fixed string matching. These computationally heavy and memory intensive tasks are used by networking equipment to inspect all packets at wire speed. The constant growth in Internet usage has made them increasingly difficult to implement at core network line speeds. Packet classification is used to sort packets into different flows by comparing their headers to a list of rules. A flow is used to decide a packet's priority and the manner in which it is processed. Fixed string matching is used to inspect a packet's payload to check if it contains any strings associated with known viruses, attacks or other harmful activities.

The contributions of this thesis towards the area of packet classification are hardware accelerators that allow packet classification to be implemented at core network line speeds when classifying packets using rulesets containing tens of thousands of rules. The hardware accelerators use modified versions of the HyperCuts packet classification algorithm. An adaptive clocking unit is also presented that dynamically adjusts the clock speed of a packet classification hardware accelerator so that its processing capacity matches the processing needs of the network traffic. This keeps dynamic power consumption to a minimum.

Contributions made towards the area of fixed string matching include a new algorithm that builds a state machine that is used to search for strings with the aid of default transition pointers. The use of default transition pointers keep memory consumption low, allowing state machines capable of searching for thousands of strings to be small enough to fit in the on-chip memory of devices such as FPGAs. A hardware accelerator is also presented that uses these state machines to search through the payloads of packets for strings at core network line speeds.

LIST OF FIGURES

Fig. 1.1. TCP/IP model showing packets being sent between end hosts through a router. 5
Fig. 1.2. Overview of the Internet architecture. 6

Fig. 2.1. Structure of rules used for packet classification. 16
Fig. 2.2. HiCuts decision tree (left) and its geometric representation (right). 19
Fig. 2.3. HyperCuts decision tree (left) and its geometric representation (right)............ 20
Fig. 2.4. Extended Grid-of-Tries with Path Compression. 21
Fig. 2.5. Recursive Flow Classification search structure. 22
Fig. 2.6. Tuple Space Search with Tuple Pruning. 23
Fig. 2.7. Memory needed for the search structures. 25
Fig. 2.8. Worst case number of memory accesses needed to classify a packet............... 26
Fig. 2.9. Energy used building the search structure. 27
Fig. 2.10. Average energy needed to classify a packet. 28
Fig. 2.11. Total number of packets classified in one second. 29
Fig. 2.12. Charging and discharging of a capacitive load. 39
Fig. 2.13. Switching characteristics of a CMOS inverter. 40
Fig. 2.14. Static vs. dynamic power. 42
Fig. 2.15. Implementation of a parallel and pipelined three input adder. 45

Fig. 3.1. Cuts performed to the root node of a decision tree. 52
Fig. 3.2. Cuts performed to the internal node of a decision tree. 53
Fig. 3.3. Traversing a decision tree to find a matching rule... 54
Fig. 3.4. Heuristics used by HyperCuts to reduce memory consumption. 55
Fig. 3.5. Region division with and without region compaction. 60
Fig. 3.6. Compacting of a region through pre-cutting.. 62
Fig. 3.7. Encoding scheme used for source and destination IP address. 63
Fig. 3.8. Layout of information needed to match a packet header to a rule.................... 64
Fig. 3.9. Architecture of cut selection logic. 65
Fig. 3.10. Layout of root/internal node when using ultra-wide memory. 66

Fig. 3.11. Layout of leaf node when using ultra-wide memory. 67
Fig. 3.12. Layout of root node cut information when using reduced width memory. 69
Fig. 3.13. Layout of root node pointers when using reduced width internal memory. 69
Fig. 3.14. Layout of internal node when using reduced width internal memory. 70
Fig. 3.15. Layout of leaf node when using reduced width internal memory. 70
Fig. 3.16. Layout of root node pointers when using reduced width external memory. ... 71
Fig. 3.17. Layout of internal node when using reduced width external memory............ 71
Fig. 3.18. Layout of leaf node when using reduced width external memory.................. 72
Fig. 3.19. Block diagram of the architecture used by the packet classification engines.. 73
Fig. 3.20. Operation of engine using ultra-wide memory words. 74
Fig. 3.21. Architecture of tree traverser using ultra-wide memory words...................... 75
Fig. 3.22. Architecture of leaf node searcher using ultra-wide memory words.............. 76
Fig. 3.23. Operation of engine using reduced width internal memory............................ 77
Fig. 3.24. Architecture of tree traverser using reduced width internal memory. 78
Fig. 3.25. Architecture of leaf node searcher using reduced width internal memory. 79
Fig. 3.26. Operation of engine using reduced width external memory........................... 80
Fig. 3.27. Architecture of leaf node searcher using reduced width external memory. 81
Fig. 3.28. Architecture of packet buffer used by packet classifiers. 84
Fig. 3.29. Architecture of sorter logic block used by packet classifiers. 85
Fig. 3.30. Architecture of hardware accelerator using ultra-wide memory words........... 86
Fig. 3.31. Architecture of hardware accelerator using reduced width memory words. ... 88
Fig. 3.32. Power consumed by packet classifiers implemented using Cyclone III.......... 95
Fig. 3.33. Power consumed by packet classifiers implemented using Stratix III. 96

Fig. 4.1. Throughput of a 24-hour trace from the CENIC HPR backbone link. 104
Fig. 4.2. Percentage of classifier idle time when classifying packets from the CENIC trace. 106
Fig. 4.3. Switching sequences with all states used... 112
Fig. 4.4. Switching sequences with selected states used. 112
Fig. 4.5. Architecture of the adaptive clocking unit.. 113
Fig. 4.6. Architecture of low power packet classifier. 115
Fig. 4.7. Power used by the ASIC implementation of the low power classifier. 118
Fig. 4.8. Power used by the Cyclone III implementation of the low power classifier... 119
Fig. 4.9. Power used by the Stratix III implementation of the low power classifier...... 120
Fig. 4.10. Throughput of the synthetic 2.5 Gbps, 10 Gbps and 40 Gbps packet traces. 122
Fig. 4.11. ASIC power usage when classifying packets from synthetic traces............. 124
Fig. 4.12. Cyclone III power usage when classifying packets from synthetic traces..... 125
Fig. 4.13. Stratix III power usage when classifying packets from synthetic traces. 126

Fig. 5.1. Aho-Corasick state machine showing transition pointers and matched states. 130
Fig. 5.2. Sequence of strings that will be traversed if text (hishersqhhe) is searched.... 131
Fig. 5.3. Use of default transition pointers to states at a depth of one. 134
Fig. 5.4. Use of default transition pointers to states at a depth of two. 135
Fig. 5.5. Use of default transition pointers to states at a depth of three. 137
Fig. 5.6. Recording a state's depth, character value and forward pointing transitions. . 139
Fig. 5.7. Recording a state's non-forward pointing transitions. 140
Fig. 5.8. Recording the strings matched if a state is entered. 141
Fig. 5.9. Memory organisation of information needed to store a state. 142
Fig. 5.10. Possible positioning of the state types in memory and their bit size. 143
Fig. 5.11. Organisation of a lookup table memory word. 144
Fig. 5.12. Architecture of a string matching block. 145
Fig. 5.13. Architecture of the string matching engine. 148
Fig. 5.14. Architecture of the string matching scheduler. 150
Fig. 5.15. Distribution of string lengths for unique strings found in Snort ruleset. 151
Fig. 5.16. Throughput of the string matchers when using different sized rulesets. 156
Fig. 5.17. Power consumed by Cyclone III implementation of the string matcher. 157
Fig. 5.18. Power consumed by Stratix III implementation of the string matcher. 158

Fig. A. 1. Power usage of ASIC low power classifier using 5,000 rules. 168
Fig. A. 2. Power usage of ASIC low power classifier using 25,000 rules. 168
Fig. A. 3. Power usage of Cyclone III low power classifier using 5,000 rules. 169
Fig. A. 4. Power usage of Stratix III low power classifier using 5,000 rules. 169
Fig. A. 5. Power usage of Stratix III low power classifier using 25,000 rules. 169

LIST OF TABLES

Table 2.1. Sample ruleset containing five rules. 18

Table 3.1. Sample ruleset containing nine rules. 51
Table 3.2. Maximum number of cuts allowed by the cutting scheme. 59
Table 3.3. FPGA resource utilisation for packet classification hardware accelerators.... 90
Table 3.4. Memory usage (bits) and worst case number of memory accesses. 92
Table 3.5. Performance comparison of packet classification hardware accelerators....... 98

Table 4.1. Statistics on packet sizes in the CENIC HPR backbone trace. 104
Table 4.2. Clock speed associated with each state... 110
Table 4.3. FPGA memory and logic utilisation of low power packet classifier. 117

Table 5.1. FPGA resource utilisation for string matching hardware accelerators. 152
Table 5.2. Reduction in number of transition pointers stored in states......................... 154
Table 5.3. Performance comparison of string matching hardware accelerators. 159

LIST OF ACRONYMS

ACL	–	Access Control List
ACU	–	Adaptive Clocking Unit
ASCII	–	American Standard Code for Information Interchange
ASIC	–	Application Specific Integrated Chip
CENIC	–	Corporation for Education Network Initiatives in California
CMOS	–	Complementary Metal–Oxide–Semiconductor
CS	–	Connecting State
DDR2	–	Double Data Rate 2
DFA	–	Deterministic Finite Automaton
DPI	–	Deep Packet Inspection
DRAM	–	Dynamic Random Access Memory
EGT-PC	–	Extended Grid-of-Tries with Path Compression
FPGA	–	Field Programmable Gate Array
FW	–	Firewall
Gbps	–	Gigabits per second
HDL	–	Hardware Description Language
HPR	–	High Performance Research
IPC	–	Internet Protocol Chain
IPTV	–	Internet Protocol Television
ISP	–	Internet Service Provider
LPM	–	Longest Prefix Match

LSB	–	Least Significant Bit
LUT	–	Lookup Table
Mbps	–	Megabits per second
Mpps	–	Million packets per second
MSB	–	Most Significant Bit
MTU	–	Maximum Transmission Unit
NAT	–	Network Address Translation
NLANR	–	National Laboratory for Applied Network Research
OC	–	Optical Carrier
OSI	–	Open System Interconnect
PE	–	Processing Element
PLL	–	Phase Lock Loop
QoS	–	Quality of Services
RAM	–	Random Access Memory
RFC	–	Recursive Flow Classification
RISC	–	Reduced Instruction Set Computer
RTL	–	Register Transfer Level
SDRAM	–	Synchronous Dynamic Random Access Memory
SRAM	–	Static Random Access Memory
TCAM	–	Ternary Content Addressable Memory
TCP	–	Transmission Control Protocol
TCP/IP	–	Transmission Control Protocol/Internet Protocol
TSMC	–	Taiwan Semiconductor Manufacturing Company
TSS	–	Tuple Space Search
UDP	–	User Datagram Protocol
VCD	–	Value Change Dump
VoIP	–	Voice over Internet Protocol
VPN	–	Virtual Private Network

PUBLICATIONS AND PATENTS ARISING FROM WORK

Published Papers

D. Bermingham, A. Kennedy, X. Wang, and B. Liu, "Architectures for the Whirlpool Hashing Algorithm," *In Proc. of the China-Ireland International Conference on Information and Communications Technologies (CIICT)*, Hangzhou, 8-19 Oct. 2006, pp.201-205.

D. Bermingham, A. Kennedy, X. Wang, and B. Liu, "A Survey of Network Processor Workloads," *In Proc. of the China-Ireland International Conference on Information and Communications Technologies (CIICT)*, Dublin, 28-29 Aug. 2007, pp.354-361.

A. Kennedy, D. Bermingham, X. Wang, and B. Liu, "Power Analysis of Packet Classification on Programmable Network Processors," *In Proc. of the IEEE International Conference on Signal Processing and Communications (ICSPC)*, Dubai, 24-27 Nov. 2007, pp.1231-1234.

A. Kennedy, X. Wang and B. Liu, "Energy Efficient Packet Classification Hardware Accelerator," *In Proc. of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Florida, 14-18 April 2008.

A. Kennedy, X. Wang, Z. Liu and B. Liu, "Frequency Scaling for Multidimensional Packet Classification," *In Proc. of the China-Ireland International Conference on Information and Communications Technologies (CIICT)*, Beijing, 26-28 Sept. 2008, pp. 383-387.

A. Kennedy, X. Wang, Z. Liu and B. Liu, "Low Power Architecture for High Speed Packet Classification," *In Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, San José, 6-7 Nov. 2008, pp. 131-140.

A. Kennedy, Z. Liu, X. Wang and B. Liu, "Multi-Engine Packet Classification Hardware Accelerator," *In Proc. of the 19th International Conference on Computer Communications and Networks (ICCCN)*, San Francisco, 2-6 Aug. 2009.

Z. Liu, A. Kennedy, O. Ormond, X. Wang, "Power-Efficient Packet Classifier for Next-Generation Routers", *European Research Consortium for Informatics and Mathematics* (ERCIM), News, No. 79, Oct. 2009.

A. Kennedy, X. Wang, Z. Liu and B. Liu, "Ultra-High Throughput String Matching for Deep Packet Inspection," *In Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, Dresden, 8-12 March 2010.

Patent Application

Patent application: Adaptive clocking system for a packet classifier. UK Patent application reference number: D07-396-27GB. Applicants: Alan Kennedy, Xiaojun Wang and Zhen Liu.

Chapter 1 - Introduction

1.1 Motivation

The increasing growth in Internet usage has been aided by its ease of access through a wide range of devices such as desktops, notebooks, netbooks, mobile phones, portable multimedia players and even watches, putting a real strain on the networking equipment needed to inspect and process the resultant traffic. A survey carried out by Internet World Stats [1] shows how this ease of access has allowed Internet penetration to reach 24.7% of the world's population as of June 2009, with the number of Internet users growing by 462% between December 2000 and June 2009. This survey also showed that 13.65% of Internet users are from the USA, which is an important statistic when it is considered that the total amount of energy used in the year 2000 by various networking devices in the USA equated to the yearly output of a typical nuclear reactor unit [2]. This would place the current amount of energy used by networking devices worldwide to be the same as the yearly output of 17 typical nuclear reactor units. Power consumption should therefore be a key concern when designing any new networking equipment for the purpose of processing the ever-increasing amount of network traffic. This is in order to slow the rapidly growing costs of running the networking equipment and to reduce their carbon footprint.

Analysis in [3] demonstrated that up to 50% of an Internet Service Provider's (ISP) maintenance costs are power related, including the electricity consumed by the routers and the corresponding cooling systems and so on. A company that manufactures power efficient networking equipment would therefore have a distinct advantage over their competitors when selling to Internet Service Providers as they could reduce their maintenance costs. Networking equipment

used to process network traffic such as high-end routers like the Cisco ASR 9010 router can consume up to 7,600 Watts, with each line card in the router consuming up to 685 Watts [4]. Due to their large integration scale and high speed, network processors deployed on a router's line card can use a large percentage of its power budget. These network processors can come in a wide range of configurations, with varying numbers of processing engines. These processing engines can run at speeds in the GHz range, consuming large amounts of power. The EZchip NP-1, for example, contains 64 processing engines [5] while the Intel IXP2800 contains 16 and has a peak power consumption of 30W [6]. Each line card on a router typically contains two network processors for ingress and egress processing, and a router can contain multiple line cards.

These network processors are used to process packets as they pass through the network, carrying out applications such as packet fragmentation and reassembly, queue management, header manipulation, encryption, forwarding, classification and pattern matching. The growing number of applications and services that need to be carried out, along with the increase in line rates, have placed the network processor under increased pressure. Relieving this pressure through the addition of extra processing capacity is not an easy task due to factors such as tight power budgets and silicon limitations. Ramping up clock speeds to gain extra performance is difficult due to physical limitations in the silicon used to manufacture these devices, while increasing the number of processing cores used to process the traffic can cause difficulty when it comes to writing the software needed to control the network processor. Both these approaches can also lead to large increases in power consumption due to the extra heat generated by increasing the clock speeds and the extra transistors needed to increase the number of processing cores.

The use of dedicated hardware accelerators designed to carry out the most computationally heavy tasks on a network processor can help to reduce power consumption while increasing processing capacity. This is because a hardware accelerator can be designed to have a smaller transistor footprint than that of the general purpose processors used as processing engines in multi-core network processors. Hardware accelerators can also process greater amounts of data than a

general purpose processor while running at much slower clock speeds as they are typically optimised to carry out a specific task. This reduction in clock speed and transistor count will lead to large savings in power consumption.

Offloading the most frequently occurring and computationally heavy tasks from a network processor's processing engines will help to prevent it from becoming a traffic bottleneck on a network, allowing for increases in achievable line rates. It will also leave the processing engines free to carry out new emerging services and protocols as they are introduced. These hardware accelerators can be placed onboard a network processor or as an external processing unit.

An explanation of the network architecture currently used by the Internet is given in Section 1.2. Section 1.3 outlines existing and emerging traffic processing bottlenecks in this architecture, which the work presented in this thesis removes through the implementation of energy efficient high throughput hardware accelerators. This section also explains the technical challenges that make the removal of these bottlenecks a difficult task. The research objectives of the thesis are stated in Section 1.4, along with the main contributions made. The thesis structure is given in Section 1.5, with Section 1.6 summarising.

1.2 Network Overview

The architecture of the communications network used by the Internet consists of end hosts, which are devices such as desktop computers, notebooks, mobile phones, etc. These end hosts communicate with each other through a web of communication mediums such as fibre optic cables, satellites and wire cables. The information sent between these end hosts is broken into pieces of data known as packets. These packets are routed through the various mediums in the communication network using devices known as routers. The communications network that these packets are sent across is governed by written standards documents known as protocols. These protocols are used to ensure the correct and efficient interoperation of the heterogeneous groups of computer networks using the Internet. They detail all aspects of communication such as the format of packets and how these packets should be handled when received. The architecture of the communications network is divided up into several distinct layers, with

each layer using one or more different protocols. A protocol suite is formed when the protocols from different layers are combined. The communications network was originally divided into seven layers before the introduction of the Internet. This was known as the Open System Interconnect (OSI) Reference Model [7]. The Internet replaced this with a five-layered model known as the Transmission Control Protocol/Internet Protocol (TCP/IP) model. Each layer is described from top to bottom as follows, where a layer provides a service to the layer above it and uses the service of the layer below it.

- Layer 5 is the highest layer and is known as the Application Layer. This layer represents the reason for communicating and is where the data being transferred is presented. It is used for applications such as file transfers, emailing or web browsing. It is the layer that the user most closely interacts with and is responsible for implementing the protocols that were carried out by the presentation and session layers. These layers were included in the OSI model but no longer exist in the TCP/IP model.
- Layer 4 is known as the Transport Layer and it is used to establish, manage and end a connection between hosts. It is also used to help make sure that packets arrive in the correct order and are error free. The transport layer is used to decide if packets should be sent using a Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). TCP can guarantee data integrity through the use of a checksum. It also guarantees delivery as it will retransmit packets until the receiver acknowledges that it has received them. This makes TCP ideal for services such as the sending of email or file transfer, where the delivery of all packets is essential. UDP also guarantees data integrity through the use of a checksum, but does not guarantee the delivery of a packet. For this reason UDP is used for sending information where the non-delivery of a few packets is not important. Examples include media applications such as Voice over Internet Protocol (VoIP) or Internet Protocol Television (IPTV).
- Layer 3 is the Internet Layer, which is used to determine how packets should be sent from the source network to the destination network through the handling of the routing. This is done by sending packets from one router to the next until the final network is reached.

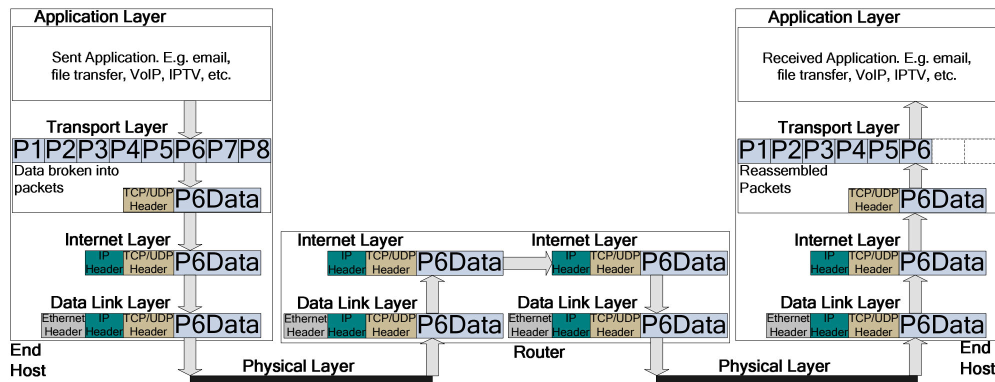


Fig. 1.1. TCP/IP model showing packets being sent between end hosts through a router.

- Layer 2 is called the Data Link Layer. It is the layer responsible for sending information between the various nodes in a communication network through the use of frames. This may involve the breaking up of large packets into multiple frames.

- Layer 1 is the lowest layer and is known as the Physical Layer. It provides electrical, optical and mechanical details about how the information should be sent across the network as bits using the various communication mediums.

An example of how the TCP/IP model can be used to send information from one end host to another is shown in Fig. 1.1. This model is based on the end-to-end design principles proposed by Saltzer et. al. [8]. They state that the majority of the communication protocols should take place at the end points of a communication system or as close to the end points as is possible. This is because the resources at the centre of the communications system will be shared by many end hosts and will therefore not have as much time to process the information being transmitted as the resources near the edge of the system, which are not so heavily shared.

The end hosts are where most of the processing on a packet occurs. This means that they require access to the full content of the packets being sent and received. This content includes the packet header and payload (the data being sent) information. A packet being sent by an end host will pass through an edge network where the packets sent by all end hosts in this network gather at an edge router. These edge networks can operate at Gigabit rates, with examples of such networks including university campuses or large company headquarters. The high rates at which these networks operate and a lack of processing capacity typically

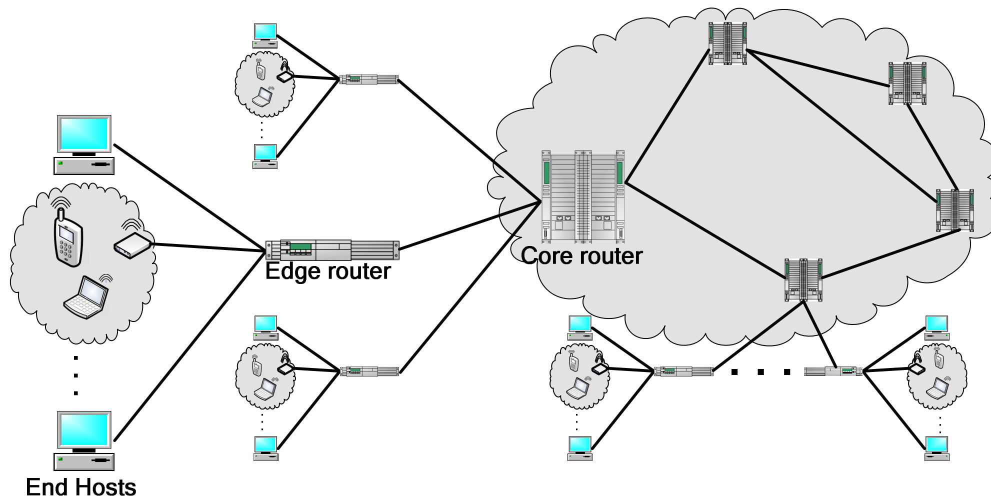


Fig. 1.2. Overview of the Internet architecture.

only give an edge router time to inspect a packet's header, allowing it to forward packets and implement vital tasks such as firewalls and Quality of Services (QoS).

A packet can be sent from the edge router to an end host in the same edge network, from an edge network to another edge network or more often to the core of the network where it is processed by core routers. The core of a network usually operates at link speeds of 10 Gigabits per second (Gbps), with 40 Gbps links also in use. At these speeds there is very little time to process a packet as it passes through a core router. A core router will typically not have time to even inspect the entire packet header and will only have time to inspect the destination IP address, allowing the router to forward a packet to its next hop. Fig. 1.2 shows the topology of the end-to-end communications network used by the Internet.

1.3 Packet Processing Bottlenecks

The work presented in this thesis centres around the design and implementation of energy efficient hardware accelerators that can relieve a network processor's processing engines of some of the most power hungry and computationally hard networking tasks. This is done to reduce power consumption and to increase a network processor's throughput, thus preventing traffic bottlenecks. A network processor has to carry out many computationally heavy tasks such as packet fragmentation and queue management. The two tasks targeted for hardware acceleration in this thesis are packet classification and fixed string matching, which is used in Deep Packet Inspection (DPI). These tasks are chosen because

they must be carried out on every packet and require search structures that use large amounts of memory, making them power hungry.

1.3.1 Packet Classification

Single-match, multi-field packet classification is the process of mapping a packet to one of a finite set of flows or categories using information from the packet's header. This information includes the source and destination IP addresses, which are matched using longest prefix matching, the source and destination port numbers, which are matched using range matching and the protocol number, which can be an exact match or wildcard. These fields are extracted from the Transport and Internet Layers of a packet's header. Packets belonging to the same flow match a predefined rule and are processed in the same way by the router's line card. The classifier will select the flow with the highest priority in the case where there are multiple rule matches. This type of packet classification usually takes place at edge routers, implementing a plethora of services such as:

- Firewalls, which are used to protect the end hosts of an edge network by blocking incoming and outgoing packets whose header information does not comply with policy. This helps to prevent harmful activity such as the spread of viruses and worms. It can also be used by an ISP to block customers from accessing prohibited websites.
- Traffic monitoring, which allows an ISP to monitor an end host's network usage, allowing it to bill appropriately.
- Traffic shaping, where some packets are delayed and others are allowed to pass through quickly. This can be used by an ISP to give priority to customers who pay more for a higher bandwidth and to allow time-sensitive traffic such as VoIP and IPTV packets to pass through quickly.
- Traffic policing, which can be used by an ISP to prevent end hosts from exceeding their maximum bandwidth limit.
- Network Address Translation (NAT), allowing multiple computers on an edge network to share a single IP address. The NAT system will rewrite the packet's header if it matches a certain flow.

- Load balancing, where large websites increase performance by running copies of their website on different servers. Packets classification is used to direct packets in a particular flow to the server with the smallest load.

The process of packet classification is an NP-hard problem, which is further complicated by the fact that all packets entering a router must be processed at wire speed. The large number of services being provided by network providers makes this problem even more difficult as rulesets containing thousands of rules are needed. Software approaches to packet classification use various algorithms [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19] which are run on the processing engines of multi-core network processors. The most common hardware approaches at high throughput packet classification include the use of power hungry memories such as Ternary Content Addressable Memory (TCAM) [20].

1.3.2 Deep Packet Inspection

Network intrusion detection/prevention systems used for the deterrence of malicious attacks depend heavily upon DPI. DPI involves searching a packet's header and payload against thousands of rules to detect a possible attack. The end-to-end architecture of the Internet means that the processing of any Application Layer data such as the packet content can only take place at end hosts and edge routers. This is because core routers do not have the processing capacity needed to inspect the entire content of a packet at wire speed. The lack of intrusion detection systems in a network leaves end hosts particularly vulnerable to attacks from malware, which is malicious software that is designed to infiltrate a computer without the owner's permission. It can be used for many purposes such as the destroying of files on a hard disk or the collection of passwords and credit card details. End hosts are also vulnerable to Internet Bots, used to carry out tasks such as the spreading of spam email.

The lack of intrusion detection systems in a network also leaves it vulnerable to viruses or worms. Slammer, the fastest spreading worm in history, infected over 75,000 hosts in only a 10-minute period [21], doubling in size every 8.5 seconds. The worm did not contain malicious content but was designed to overload a network, slowing down Internet speeds and even causing the loss of connection

for some end hosts. Another worm that caused mass damage by Denial of Service attacks was CodeRed, infecting 359,000 hosts in 14 hours [22]. With viruses/worms spreading at these speeds it would be unrealistic to expect the end hosts of a network to update their systems to new threats due to the slow time that it would take to react to the rapid attack. There is also the high cost in both the maintenance and lost work time due to updating the system.

The rules used for DPI in an intrusion detection system such as Snort [23] consist of two parts. The first part is a *header rule*, which involves performing 5-tuple multi-match packet classification on a packet's header. Multi-match packet classification differs from single-match packet classification described in Section 1.3.1 in that it will return all matching rules rather than the rule with the highest priority. The second part is a *content rule*, where a specific string or strings must be searched for in a packet's payload at given locations. Research in [24] shows that, for Snort, the fraction of time that network intrusion detection spends finding these strings on real traces is between 40-70%, using 60-80% of the instructions executed. These strings can be searched for using regular expression matching, fixed string matching, or both. The area of multi-match packet classification contains many solutions [25, 26, 27], with hardware accelerators reporting throughputs of up to 10 Gbps. There has also been much research done in the area of regular expression matching [28, 29, 30, 31, 32, 33, 34, 35], with implementations reporting throughputs of up to 5 Gbps.

The main aim of this thesis is to design hardware accelerators for the computationally heavy tasks of single-match, multi-field packet classification and fixed string matching. The work presented in this thesis is not therefore concerned with the problems of multi-match classification and regular expression matching, which are required to fully implement DPI. Any reference to packet classification in future sections will refer to single-match, multi-field packet classification, while any reference to string matching will refer to fixed string matching.

1.3.3 Technical Challenges

There are many challenges when implementing energy efficient packet classification and string matching hardware accelerators. These problems include

the low amount of processing time available to process packets and the large amounts of memory needed to store search structures. It is not possible to process packets at core network line speeds, which can reach 40 Gbps, by increasing clock speeds alone. Hardware accelerators designed to meet these speeds would need to operate in the GHz range if a single processing engine was used. These speeds are not possible on current state of the art devices such as Field Programmable Gate Arrays (FPGA), which typically run at a few hundred MHz. Running a hardware accelerator at these speeds would also have massive power implications due to large dynamic power consumption. It is therefore necessary to design more optimized hardware accelerators capable of processing multiple packets in parallel.

The search structures that these hardware accelerators use must be as compact as possible, using up only small amounts of memory. This is because devices used for implementing hardware accelerators, such as high-end FPGAs, do not contain more than a few MB of internal memory. It is important that search structures should be able to fit inside this internal memory to prevent the need for external memory. The use of external memory would drastically decrease the performance of a hardware accelerator, while adding extra power consumption. Specific packet processing tasks also have their own unique technical challenges:

- Currently packet classification is most commonly implemented on edge routers, where line rates do not typically exceed speeds of a few Gbps and rulesets do not usually contain more than a thousand rules [12, 36]. It is anticipated, however, that these rulesets will grow to contain tens of thousands of rules as services move into the network core [36]. This means that any new hardware accelerators designed for packet classification should be able to classify packets for rulesets containing tens of thousands of rules at line speeds in excess of 40 Gbps. At these speeds a classifier must be able to classify a packet in less than 8 ns. This is in order to achieve a maximum throughput in excess of 125 Million packets per second (Mpps) in the worst case when 40 byte packets arrive back-to-back.

- One of the most computationally heavy tasks in networking is the task of searching for strings in a packet's payload. This is because rulesets used for DPI such as Snort will typically contain several thousand strings that must be

searched for at wire speed. These strings can come in a variety of lengths, ranging from a few bytes to a couple of hundred bytes. Any hardware accelerator implementing string matching must be able to search for these strings at a fixed rate to guarantee a specific bandwidth, regardless of the string length. This will leave as little as 0.2 ns to inspect each byte of a packet as line rates reach 40 Gbps.

1.4 Contributions

As previously mentioned, the main focus of this thesis is on the design of high throughput and energy efficient hardware accelerators for packet classification and string matching. The contributions in these areas are described in detail in Chapters 3, 4 and 5. These contributions are summarised below.

Packet Classification

The contributions towards the field of packet classification include new multi-engine hardware accelerator architectures capable of classifying packets at line speeds in excess of 40 Gbps, while using rulesets that contain tens of thousands of rules. These hardware accelerator architectures allow packet classification to be used at the core of the network, helping to improve security. They implement modified versions of the HyperCuts [10] packet classification algorithm, which breaks a ruleset into different groups, with each group containing a small number of rules that can be searched linearly. A decision tree is used to guide a packet based on its header values to the correct group to be searched. The architectures are divided into two different types, with one type using ultra-wide memory words, making it ideally suited to classifying packets for rulesets that contain many wildcard rules. This is because the ultra-wide memory words can be used to store a large number of rules that can be retrieved from memory and searched in a single clock cycle. The number of rules in each group can therefore be quite large, which is ideal for rulesets containing many wildcard rules as they are hard to break up into small groups.

A second type of hardware accelerator is also presented that uses reduced width memory words, allowing for higher clock speeds and throughputs. It is ideally suited to rulesets that do not contain a large number of wildcard rules. This is

because rulesets need to be divided into groups that contain only a small number of rules, due to the fact that the narrow memory words can only search a couple of rules on each clock cycle. All architectures use multiple packet classification engines, which work in parallel using a shared memory. The use of multiple engines allows for the option of breaking problem rulesets containing many wildcard rules into different groups, with a separate decision tree built for each group. Each decision tree can then be searched in parallel using the multiple packet classification engines. The splitting of problem rulesets can help to improve storage efficiency and reduce the number of clock cycles needed to classify a packet. This is because rules with wildcard fields in the same location can be grouped together, allowing for better cutting efficiency as the non-wildcard ranges can be used to split the rules into small groups that can be easily searched.

Another contribution to the field of packet classification is an adaptive clocking unit designed specifically for use with packet classification hardware accelerators. The adaptive clocking unit dynamically changes the clock frequency of the packet classification hardware accelerator to match fluctuations in traffic on a router's line card. It does this with the help of a scheme developed to keep clock frequencies at the lowest speed capable of servicing the line card, while keeping frequency switches to a minimum. Line rates are monitored by capturing the fields from a packet's header needed for packet classification in a small buffer and using the number of packets buffered to decide the appropriate clock frequency. This scheme has been tested extensively using real packet traces, with simulation results showing that power savings of between 14-88% can be made when using the adaptive clocking unit rather than a fixed clock speed.

String Matching

The main contributions to the field of string matching are a new multi-pattern matching algorithm and a hardware accelerator that can search for the fixed strings contained within a DPI ruleset at a guaranteed rate of one character per cycle, independent of the number of strings or their length. The algorithm is based on the Aho-Corasick [37] string matching algorithm, with the modifications made resulting in a memory reduction of over 98% on strings tested from the Snort ruleset. This allows the search structures needed for identifying thousands of

strings to be small enough to fit in the on-chip memory of an FPGA. Combined with a simple architecture for hardware, this leads to high throughput and low power consumption. The hardware implementation uses multiple string matching engines working in parallel to search through packets. It can reach a throughput of over 40 Gbps when implemented on a Stratix III FPGA and over 10 Gbps when implemented on the low power Cyclone III FPGA.

1.5 Thesis Organisation

The remainder of this thesis is organised as follows. Chapter 2 gives background information into the area of packet classification, explaining the structure of the rulesets used to classify packets. It then gives an overview of the most popular algorithms used for packet classification. An extensive performance analysis of these algorithms is then carried out in order to identify the algorithms most suitable for hardware acceleration. A description of the Snort ruleset used for DPI is given next, followed by an overview of the most effective techniques employed for string matching. An explanation of the hardware platforms that can be used to speed up packet classification and string matching is also given. This is followed by an explanation of the main causes of power consumption in these hardware platforms and an analysis of low power design techniques that can be used to reduce power consumption.

Chapter 3 describes the architecture of the hardware accelerators designed for packet classification, giving detailed descriptions of the cutting schemes used to build the search structures, and their memory organisation. Performance results for the hardware accelerators are then given, showing their power consumption, throughput and memory usage. A comparison with state of the art commercial approaches and prior art is also given.

Chapter 4 explains the motivation for the use of frequency scaling and presents the results of an analysis on the bandwidth utilisation of real backbone traces. Details on the frequency switching scheme developed are then given, along with an explanation of the adaptive clocking unit architecture. The power savings made by using the adaptive clocking unit to clock a packet classifier rather than a fixed clock speed are then presented.

Chapter 5 presents the new multi-pattern matching algorithm and hardware accelerator. It also gives details on how the search structure built by this algorithm can be stored in a memory efficient manner. Details of the hardware accelerator architecture are also given, along with performance results. These performance results show the memory reductions made by the new algorithm, throughput of the hardware accelerator, power consumption and a comparison of the work with prior art.

Chapter 6 summarises the results achieved in Chapters 3, 4 and 5. It also gives directions for future research ideas.

1.6 Summary

A real strain has been put on the networking devices used to process packets as they pass through a network. This is due to the ever-increasing growth in Internet usage and the rising number of applications that need to be provided at the core of a network to ensure QoS and the protection of end hosts from security threats. The increased workload has lead to a large increase in the amount of power used by networking equipment. Two of the applications that need to be provided by networking devices are the computationally heavy tasks of packet classification and string matching used to implement DPI. These applications have to process packets at wire speed, which is not an easy task, with line rates reaching up to 40 Gbps. The work in this thesis helps to remove these packet processing bottlenecks through the implementation of two energy efficient high throughput hardware accelerators for packet classification and one for string matching. An adaptive clocking unit is also presented that dynamically adjusts the clock speed to a packet classifier so that its processing capacity matches the processing needs of the network traffic on a router's line card, reducing power consumption.

Chapter 2 - Background

2.1 Introduction

The areas of packet classification and string matching are complex and challenging fields with a wide range of solutions. This chapter gives a technical overview of these fields in order to provide context for the research presented in the following chapters. It begins with an explanation of the rulesets used for packet classification. This is followed by a detailed analysis of five of the most popular packet classification algorithms. These algorithms are implemented in C code and simulated on a SA1100-StrongARM Reduced Instruction Set Computer (RISC) processor similar to the type used as processing cores in many of today's programmable network processors. Their performance is compared in terms of the amount of memory needed to store their search structure, worst case number of memory accesses needed to classify a packet, energy used building the search structure, average energy needed to classify a packet and their average throughput. The algorithms are tested using rulesets of different sizes. These tests are carried out in order to determine which algorithm would be best suited to hardware acceleration and the ability of these algorithms to scale, allowing for the handling of rulesets containing tens of thousands of rules.

An explanation of the rulesets used in DPI is then given, along with a brief description of some of the most commonly used approaches at implementing the task of string matching, which is needed for DPI. A description of the hardware platforms that can be used to implement hardware accelerators aimed at packet classification and string matching is also given, stating their advantages and disadvantages. The types of power dissipation that can occur in digital circuitry and their causes are also explained, as well as a method for power benchmarking. Methods for the design of hardware accelerator architectures with reduced power consumption are also discussed.

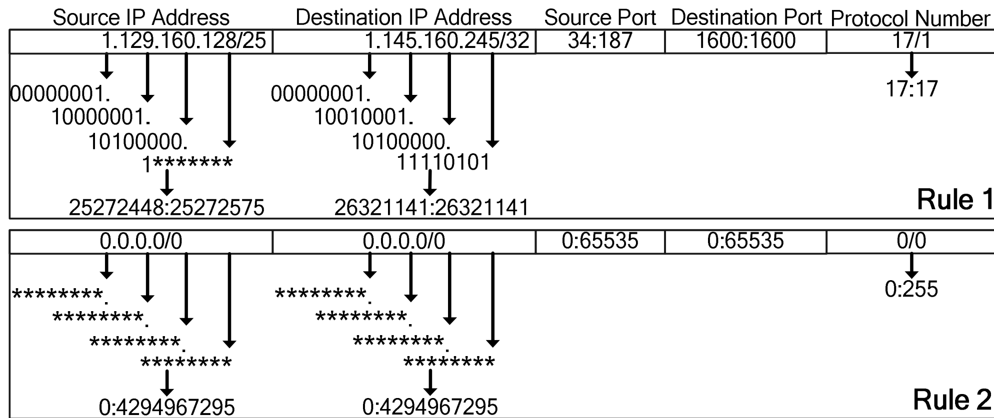


Fig. 2.1. Structure of rules used for packet classification.

2.2 Packet Classification Rulesets

A packet classification ruleset is used to sort packets into flows, with a flow obeying at least one rule in a ruleset. The fields most commonly used in a packet header to perform multi-dimension packet classification are the source IP address, destination IP address, protocol number (all taken from the Internet Layer of the TCP/IP model), source port and destination port (both taken from the Transport Layer of the TCP/IP model). Packet classifiers that only use these fields to classify packets are stateless, which means that they treat each packet in isolation and have no memory of previous packets. This is in contrast to stateful packet classifiers which keep track of the state of network connections.

Fig. 2.1 shows an example of two rules, with rule 1 showing the format of a typical rule and rule 2 showing the format of a rule where all fields are wildcards, meaning that all packet headers would return a match. The source and destination IP addresses are 32-bit numbers that are matched using prefix matching. Each IP address is usually stored in a rule using four 8-bit numbers and a 6-bit mask. These four 8-bit numbers are concatenated to form the 32-bit IP address. The mask is used to specify the number of Most Significant Bits (MSB) that must be an exact match to the corresponding bits in the packet header to record a match. The remaining Least Significant Bits (LSB) are wildcard bits, meaning that the corresponding bits in the packet header can be any value and still record a match.

The source and destination port numbers use range matching, with each port number in a rule stored using two 16-bit numbers, representing the minimum and maximum range values. A packet will record a match for these fields if its port

numbers are within these ranges. The final field used is the protocol number, which can be an exact match or wildcard. Each rule will require eight bits to specify the protocol number and one bit to state if the corresponding field in the packet header must match exactly or is a wildcard, meaning that any value will return a match.

Due to security and confidentiality issues it is difficult to obtain access to real rulesets used by an ISP. A problem with the use of rulesets used by a specific ISP in the testing and evaluation of new packet classification algorithms and hardware accelerators is that it can be difficult to compare the performance of new research to that of prior art. This is due to the possibility of large differences in the structure of the rulesets and packet headers used in testing. For these reasons ClassBench [36] the de facto suite of tools used for the benchmarking of packet classification algorithms and devices is employed here. The ClassBench suite of tools consists of a ruleset generator which is used to create synthetic rulesets that accurately model the characteristics of real rulesets. The suite of tools also contains a trace generator which creates packet headers that match the rules contained within the synthetic rulesets created by the ruleset generator.

The ruleset generator creates Access Control List (ACL), Firewall (FW) and Internet Protocol Chain (IPC) rulesets. ACL rulesets are used for security, Virtual Private Networks (VPN), and Network Address Translation (NAT) rules for firewalls and routers. FW rulesets are used for specifying security rules for firewalls and IPC rulesets are used for security, VPN and NAT rules for software-based systems. The ruleset generator uses an input parameter file known as a seed filter set that describes the characteristics of the type of ruleset to be generated. This is used to create a ruleset in conjunction with settings specified by the user such as the number of rules to be created, scope of the ruleset (states how specific the rule values should be) and smoothness of rulesets (used to introduce new address aggregates when creating large rulesets).

2.3 Analysis of Software Approaches to Packet Classification

The most basic method for implementing packet classification is to perform a linear search of all rules stored within a ruleset. To do this the rules are stored in order of decreasing priority. The rules are compared sequentially to the appropriate

Table 2.1. Sample ruleset containing five rules.

RuleID	S. IP	D. IP	S. Port	D. Port	Protocol	Action
R_1	111*	010*	78-78	230-702	UDP	ACT_1
R_2	111*	1***	0-2000	10-10	UDP	ACT_2
R_3	1***	101*	30-80	0-65535	TCP	ACT_3
R_4	10**	000*	0-65535	960-990	TCP	ACT_4
R_5	00**	101*	0-65535	800-811	TCP	ACT_5

header fields of an incoming packet until a match takes place. This method of packet classification will result in a storage efficient search structure but will have a high search time, making it unsuitable for large rulesets. In order to reduce the search time many algorithms have been developed to carry out packet classification. These algorithms spend time pre-processing the ruleset guided by various heuristics in order to build a search structure that reduces search time at the cost of increased memory consumption. The goal of all these algorithms is to keep the memory used to store the search structure and the number of memory accesses required to match a packet to a rule in the ruleset as low as possible. The algorithms can be divided into three distinct categories. These are decision tree-based [9, 10, 11, 15, 18] decomposition-based [12, 13] and hash-based [16].

The following section explains five of the most commonly used algorithms when it comes to implementing packet classifiers in software. These algorithms have been implemented in C code, with their performance compared against each other. This is done in order to find out which algorithms scale well in terms of memory usage and throughput when large rulesets are used. It was also done to figure out which algorithms might benefit most from hardware acceleration. Table 2.1 shows a simple ruleset containing five rules and the action that must be taken if a specific rule is returned as a correct match. The purpose of this ruleset is to aid in the explanation of the algorithms described in the following section. The number of bits representing the source and destination IP addresses has been reduced from 32 to 4 bits to aid the explanation.

2.3.1 Algorithmic Approaches

Hierarchical Intelligent Cuttings (HiCuts)

HiCuts by Gupta and McKeown [9] is a decision tree-based algorithm that allows incremental updates to a ruleset. It takes a geometric view of packet classification

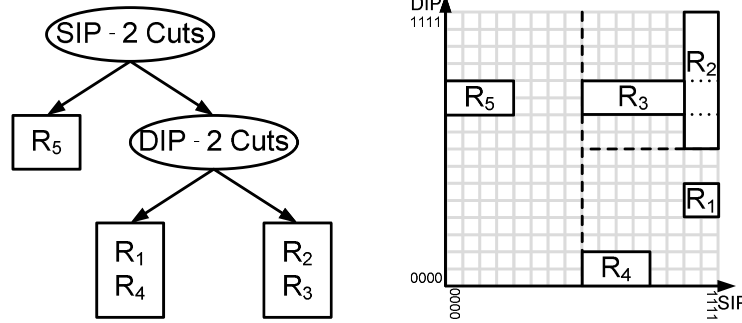


Fig. 2.2. HiCuts decision tree (left) and its geometric representation (right).

by considering each rule in a ruleset as a hypercube in hyperspace, defined by the *F* fields of a packet's header. The algorithm constructs the decision tree by recursively cutting the hyperspace one dimension at a time into sub-regions. These sub-regions will contain the rules whose hypercube overlap. Each cut along a dimension will increase the number of sub-regions, with each sub-region containing fewer rules. The algorithm will keep cutting into the hyperspace until none of the sub-regions contain more rules than is specified by a predetermined number called *binth*.

Fig. 2.2 shows a decision tree built from the ruleset in Table 2.1 where a *binth* value of two is used. It also includes a geometric representation of the source and destination IP addresses, showing the cuts made to create the decision tree. The source IP address is selected to cut the root node in two, resulting in two child nodes of which one exceeds the *binth* value. The node exceeding *binth* value is split in two using the destination IP address, with the number of rules in both child nodes equalling the predetermined *binth* value. The more cuts performed to an internal node (represented by an ellipse in Fig. 2.2), the fatter and shorter the decision tree. A fatter decision tree will require fewer memory accesses to classify a packet as less internal nodes will need to be traversed. Too many cuts, however, will result in an unacceptable amount of memory needed to store the decision tree. For that reason the number of cuts that can be performed on a dimension at an internal or root node is limited using a set of rules and a user defined variable known as *spfac*.

Each time a packet arrives the tree is traversed from the root node until a leaf node (represented by a rectangle in Fig. 2.2) is found. This leaf node will store a small

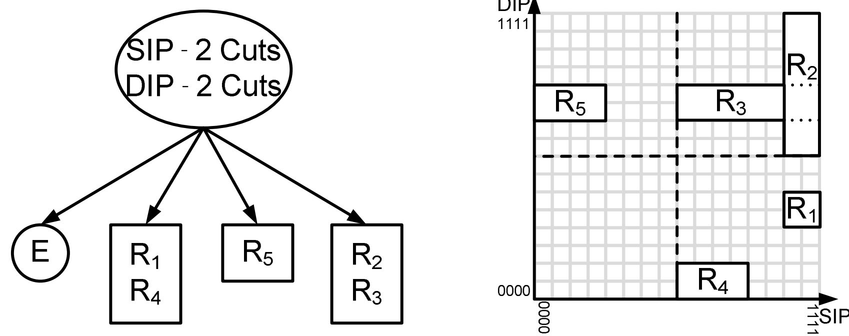


Fig. 2.3. HyperCuts decision tree (left) and its geometric representation (right).

number of rules limited by the *binth* value. Once a leaf node is reached, a short linear search of the rules contained within it is performed to find the matching rule. HiCuts uses heuristics to reduce memory usage, such as the merging of identical nodes to avoid replicated storage and the removal of rules from a leaf node that can never be matched as they are covered in that leaf node by a rule with a higher priority.

Multidimensional Cutting (HyperCuts)

HyperCuts by Singh et al [10] is a modification of the HiCuts algorithm that also allows incremental updates. The main difference between it and HiCuts is that it recursively cuts the hyperspace into sub-regions by performing cuts on multiple dimensions at a time. Fig. 2.3 shows an example of a decision tree built from the ruleset in Table 2.1. It also includes a geometric representation of the source and destination IP addresses, showing the cuts made to create the decision tree. The source and destination IP addresses are both cut in two, resulting in one empty node (represented by a circle) and three leaf nodes. All child nodes conform to the *binth* value, meaning that no more cutting is required. HyperCuts acts like HiCuts if only one dimension is chosen for cutting. The algorithm also limits the number of cuts that can be performed to an internal or root node to prevent excess memory usage, using a set of rules and a user defined variable known as *spfac*.

HyperCuts also takes advantage of extra heuristics that exploit the structure of the classifier such as region compaction, which allows for more efficient cutting of a dimension as it only cuts the region covered by the rules rather than the full region. It also pushes common rule subsets upwards to avoid the replicated storage

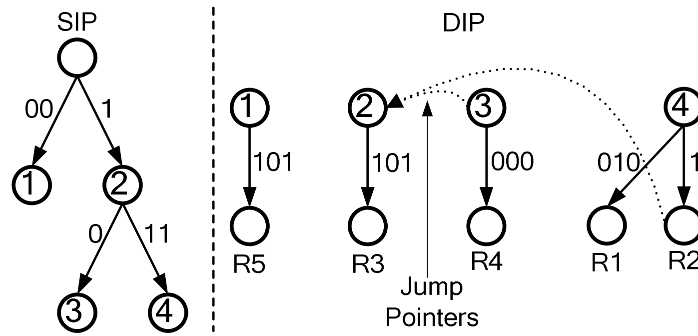


Fig. 2.4. Extended Grid-of-Tries with Path Compression.

of rules by storing rules common to all child nodes in their parent node. A packet is classified in the same manner as the HiCuts algorithm, with a packet traversing the decision tree by using the same cutting sequence on the header used to create the decision tree until a leaf node is found, where a linear search of the rules within it takes place.

Extended Grid-of-Tries with Path Compression (EGT-PC)

EGT-PC by Baboescu et al [15] is another decision tree-based algorithm that allows incremental updates. In EGT-PC a path compressed trie is first created from the prefixes in the ruleset's first dimension. Each node in this trie, which represents a valid prefix *P* in the first dimension, will contain a pointer to another path compressed trie made up of all the prefixes from the second dimension whose first dimension prefix is equal to *P*. Each node in the second dimension trie corresponding to a valid prefix in this dimension will contain a list of all the rules that match the prefixes of the first and second dimension nodes. This means that a rule can only occur in one position. In order to avoid back tracking, all failure points in the second dimension tries contain a jump pointer, which points to the next possible second dimension trie that could contain a matching rule. Fig. 2.4 shows the search structure built from the rules in Table 2.1.

The search algorithm works by first performing a Longest Prefix Match (LPM) on the first dimension trie. The resulting pointer is then followed to a second dimension trie. A LPM is then carried out on this trie to find nodes containing matching rules. Each time there is a failure or the end of a second dimension trie is reached, a jump pointer is followed. This is continued until a node is reached

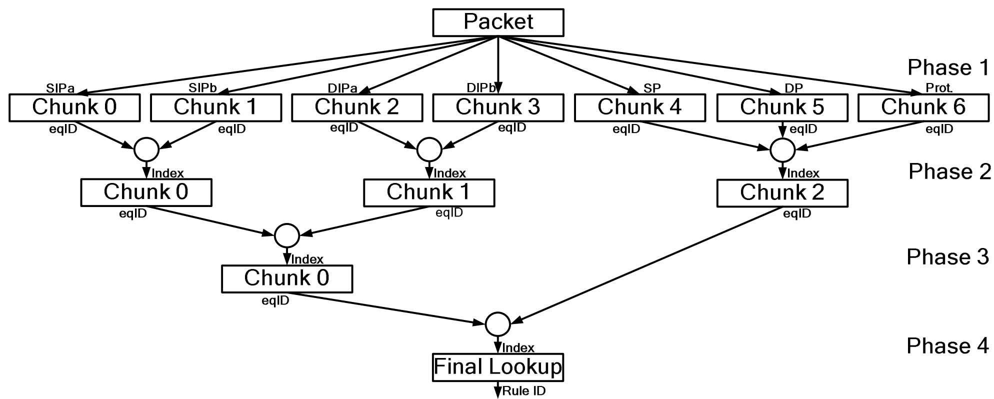


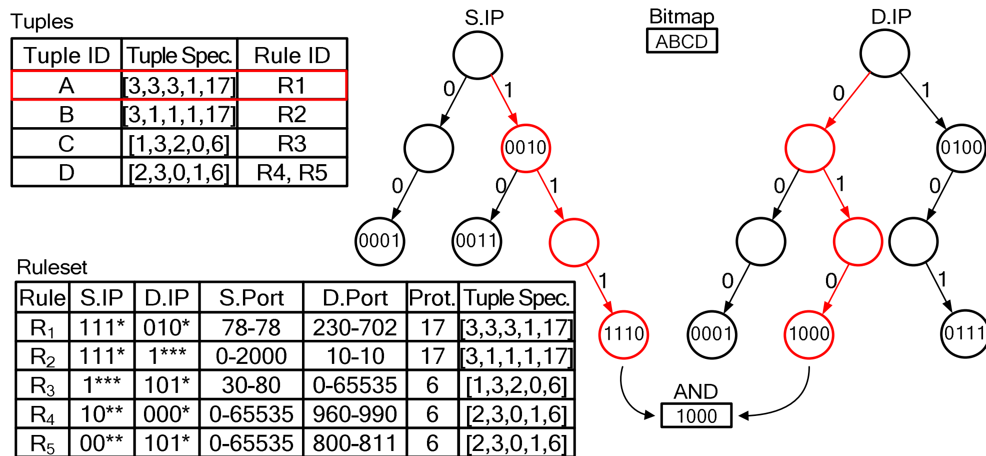
Fig. 2.5. Recursive Flow Classification search structure.

that contains no jump pointer. All matching rules along the way are recorded, with a small linear search of these rules carried out at the end.

Recursive Flow Classification (RFC)

RFC by Gupta and McKeown [12] is a decomposition-based algorithm that classifies packets at high throughput rates using lookup tables placed across multiple phases. It does this at the cost of a long pre-processing time when building these tables, high memory consumption and an inability to allow incremental ruleset updates. It uses the fields from a packet's header as indexes to access direct lookup tables in the first phase. These lookup tables are built from the corresponding fields of the rules in the ruleset. The size of each lookup table in this phase will be 2^n, where n is the number of bits in a given field. The source IP and destination IP address are usually split into 16-bit chunks to prevent their lookup tables having excessive memory consumption. This means that each IP address requires two lookup tables in the first phase, with the remaining fields requiring one each.

The lookup tables in the first phase are accessed in parallel, returning pre-processed *eqIDs*. These *eqIDs* represent and are smaller than the indexes used to access the lookup tables. The indexes for performing lookups on tables in the next phase are formed by combing the *eqIDs* from the previous phase. The final phase contains one lookup table, with the value returned from this being the matching rule number. This is possible because of the way that the lookup tables are constructed. Fig. 2.5 shows the configuration of the twelve lookup tables in the implementation used here and how they are spread across four phases.



Tuples

Tuple ID	Tuple Spec.	Rule ID
A	[3,3,3,1,17]	R1
B	[3,1,1,1,17]	R2
C	[1,3,2,0,6]	R3
D	[2,3,0,1,6]	R4, R5

Ruleset

Rule	S.IP	D.IP	S.Port	D.Port	Prot.	Tuple Spec.
R_1	111*	010*	78-78	230-702	17	[3,3,3,1,17]
R_2	111*	1***	0-2000	10-10	17	[3,1,1,1,17]
R_3	1***	101*	30-80	0-65535	6	[1,3,2,0,6]
R_4	10**	000*	0-65535	960-990	6	[2,3,0,1,6]
R_5	00**	101*	0-65535	800-811	6	[2,3,0,1,6]

Fig. 2.6. Tuple Space Search with Tuple Pruning.

Tuple Space Search (TSS) with Tuple Pruning

TSS by Srinivasan et al [16] is a hash-based algorithm that supports incremental updates. All rules are divided into groups called tuples, with rules that map to a particular tuple having the same prefix length for the source and destination IP addresses. Their source and destination port numbers will either be a wildcard or the same nesting level inside the port range. Protocol values will either be wildcard or a specific value. The nesting level of a port will help to distinguish the tuple group the rule belongs to but will not help to separate the rule from other rules within a tuple group. For this reason each port address will have a *RangeId*, which notes a port's position inside its nesting level. A packet's port number is usually converted to its *RangeId* using a 65KB direct lookup table.

A hash key is made for a tuple group by using its tuple specification e.g. (3, 1, 1, 2, 17) to pick out the appropriate bits from a packet's source and destination IP address, *RangeIds* (found using the port numbers) and protocol number. All rules belonging to tuple T are stored in Hashtable (T). A probe of a tuple T is carried out using the hash key created, with only one memory access needed for each tuple to determine if it contains a matched rule. The algorithm is motivated by the fact that a linear search through all tuples will be smaller than a search through all rules.

The number of tuples that need to be searched is further reduced through tuple pruning. Tuple pruning involves creating LPM tries, which are usually made from the source and destination IP addresses. Each node in a trie that represents a valid prefix will contain a bitmap, with each set bit in the bitmap indicating a particular

tuple that could contain a rule match. The deepest bitmap reached is given as the result of a LPM. An AND operation is performed on the bitmaps returned from the source and destination IP address tries in order to figure out which tuples need to be searched. Fig. 2.6 shows the tuple groups and LPM tries created from the rules in Table 2.1. It also shows how tuple pruning can be used to reduce the number of tuples that need to be searched when seeking a rule match for a packet with a source IP address of 1110 and a destination IP address of 0101. Only one tuple group will need to be searched in this case instead of a worst case of four.

2.3.2 Simulation Framework

The five packet classification algorithms described have been simulated on a SA1100-StrongARM processor as it is similar in architecture to the type of processor used by multi-core network processors. The simulator Sim-Panalyzer [38] was used to do this as it is able to estimate a program's run time and average power consumption. This allows measurements of the amount of energy needed to build the search structures, average energy needed to classify a packet and throughput to be taken. Sim-Panalyzer is an infrastructure for microarchitectural power simulation implemented on top of "Sim-Outorder", a component within the SimpleScalar simulator. It simulates the execution of instructions at the level of individual cycles, keeping track of power changes across cycles.

The simulator consists of several distinct components. These components are cache power models, datapath and execution unit power models, clock tree power models and I/O power models. It is worth noting that the simulator does not take into account the amount of power that would be used by the external memory needed to save the search structures created by the algorithms. Sim-Panalyzer was configured to simulate the SA1100-StrongARM processor running at a clock speed of 200 Mhz, while operating at 1.8 V using 0.18μm technology.

The code written for the five algorithms has been tested extensively using ACL rulesets and their corresponding packet traces, which were generated using ClassBench. Gupta and McKeown carried out an extensive study of rulesets [12] and found that only 0.7% of the rulesets that they examined contained over 1,000 rules and that none contained more than 2,000 rules. These findings were backed up by analysis of real rulesets by Taylor and Turner [36] which found that the

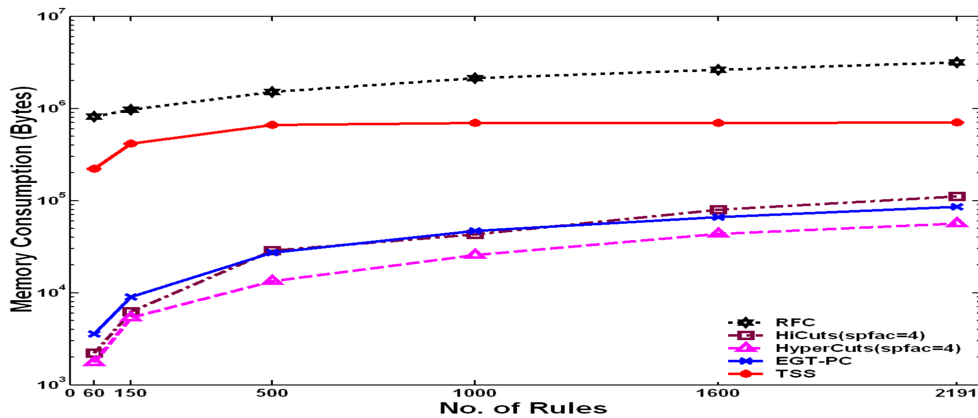


Fig. 2.7. Memory needed for the search structures.

rulesets in edge routers do not typically contain more than a thousand rules. With these points in mind it was decided that the use of rulesets with just over 2,000 rules would be enough to extensively test the algorithms. The rulesets used in testing contained between 60-2,191 rules.

2.3.3 Performance Results

The first results presented are the amount of memory needed to save the search structures built by the five algorithms. This is followed by the worst case number of memory accesses needed to classify a packet. These results have been widely analysed by prior art [10, 15, 39]. The results in prior art, however, never compare results such as the energy usage of these algorithms and their throughput, which are important factors. The results presented here cover these areas extensively, showing the energy used by the algorithms during the building of the search structure, average energy needed to classify a packet and the average number of packets that can be processed per second when running the algorithms on a SA11000-StrongARM processor.

Memory Consumption

The results in Fig. 2.7 show the memory needed to store the search structures built by the five algorithms tested. It can be seen that the worst performing algorithm in this area is RFC, needing over 3 MB of memory when 2,191 rules are used. This is due to the large amount of memory that is required to store the direct lookup tables. The second worst performing algorithm tested in this area is TSS. This is because of the large 65 KB direct lookup tables needed for converting the port numbers to *RangeIds* and the hash tables used by the tuple groups to store the

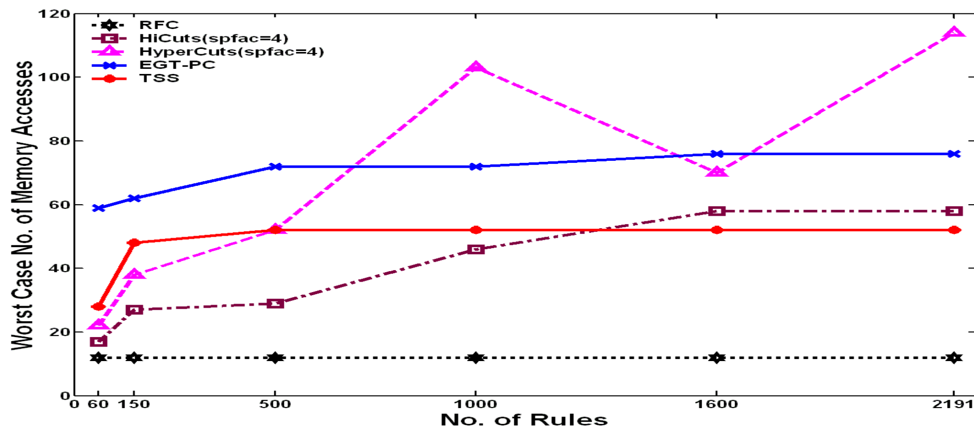


Fig. 2.8. Worst case number of memory accesses needed to classify a packet.

matching rule numbers. HyperCuts performed best over the full range of rules, only needing between 1.7-56 KB of memory to store the search structures built for the rulesets containing between 60-2,191 rules. This is due to the simplicity of its search structure. The EGT-PC and HiCuts algorithms also performed well as they have a search structure similar to that used by HyperCuts.

Worst Case Number of Memory Accesses

Fig. 2.8 shows the worst case number of memory accesses needed to classify a packet, with RFC this time being by far the best performer, needing only twelve memory accesses to classify a packet for all sized rulesets. This is possible because RFC uses direct lookup tables, which means that the number of memory accesses will always be constant no matter how many rules are used. The TSS algorithm levelled out at a worst case number of memory accesses of 52 after 500 rules. This is due to the fact that the number of distinct tuple specifications stopped growing after this point, meaning that the LPM trees never got deeper and the number of tuple groups to be searched never grew. This did not, however, mean that the TSS algorithm scaled well to large rulesets as the chances of hash collisions increased significantly as the number of rules increased.

The worst performing decision tree-based algorithm was EGT-PC, due to the fact that it had one of the most complex search structures. This is because each packet has to perform a LPM on a decision tree built from the source IP address and on multiple decision trees built using the destination IP address. HiCuts was the best performing decision tree-based algorithm, outperforming HyperCuts. This was due to the fact that HyperCuts needed to access extra information when traversing

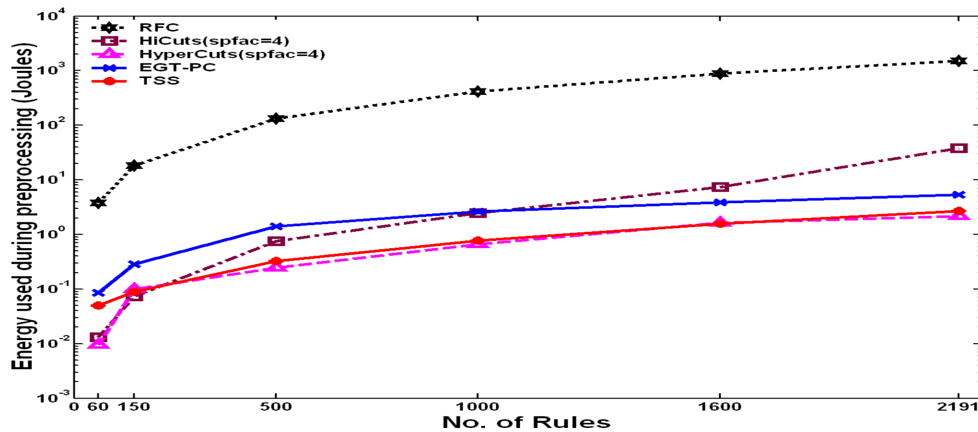


Fig. 2.9. Energy used building the search structure.

the decision tree. This information includes multiple dimensions, which may need to be cut, and the minimum and maximum range values for these dimensions when using the Region Compaction heuristic. HyperCuts was also very restrictive in the number of cuts it allowed to an internal node, meaning that deeper decision trees were needed.

It can be seen from looking at Fig. 2.8 that for HyperCuts the worst case number of memory accesses needed to classify a packet is 103 when 1000 rules are used and only 70 when 1600 rules are used. This dip is due to the rule HyperCuts uses to limit the number of cuts that can be made to an internal node when building a decision tree. The number of cuts allowed to an internal node is proportional to the number of rules it contains. The decision tree built for the ruleset with 1600 rules allows more cuts to the root node than the decision tree built for the ruleset 1000. For these particular rules the result is that the decision tree built for the rulesets with 1000 rules will be deeper than the decision tree built for the ruleset with 1600 rules. That is why in this example the worst case number of memory accesses needed to classify a packet is smaller for the bigger ruleset.

Energy Used Building the Search Structure

The amount of energy and time used when building a search structure are directly related, with these metrics not of much importance to algorithms that support incremental updates. This is because their search structures will not need to be rebuilt very often. These metrics are, however, of great importance to algorithms that do not support incremental ruleset updates, as search structures will need to be rebuilt regularly. Fig. 2.9 shows the amount of energy used when building the

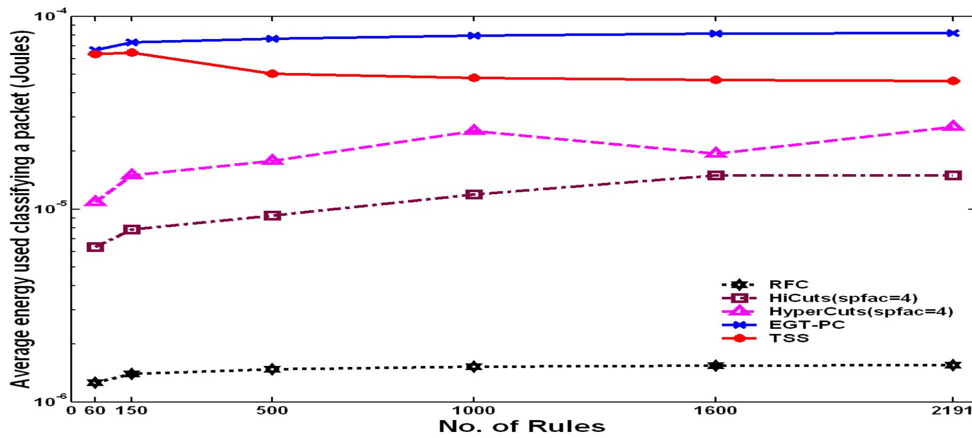


Fig. 2.10. Average energy needed to classify a packet.

search structures for the five algorithms tested. Looking at Fig. 2.7 it can be seen that the amount of memory needed to save these search structures is also proportional to the amount of energy used when building them. This means that an algorithm with low memory consumption may also use a low amount of energy and require reduced processing time when building its search structure.

The worst performing algorithm by far is RFC as it uses 1,512 Joules of energy to build its search structure when using 2,191 rules. This is high when compared to HyperCuts, the best performing algorithm, which only requires 2.7 Joules. RFC shows such poor performance due to the complexity involved in building the many large lookup tables that it needs. The EGT-PC and TSS algorithms also scale well when it comes to the amount of energy used when building their search structure for different sized rulesets. HiCuts performs slightly poorer as the rulesets become large, using 37.9 Joules of energy to build its search structure for 2,191 rules. This should not be a problem, however, as HiCuts supports fast incremental updates to the ruleset, meaning that the search structure will not have to be rebuilt regularly.

Average Energy Needed to Classify a Packet

The important metric of the average energy needed to classify a packet can be seen in Fig. 2.10. For the algorithms that support incremental updates, this graph will represent the majority of the energy used during packet classification. The algorithm that uses the least amount of energy when classifying a packet is RFC, using on average 1.46 μJ. This is due to the simplicity of its search structure,

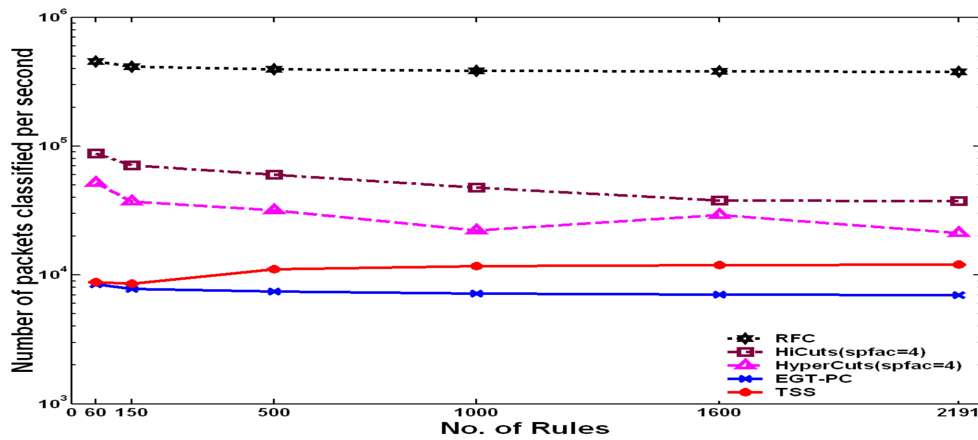


Fig. 2.11. Total number of packets classified in one second.

which only requires twelve memory lookups and a few multiplications to classify each packet. The worst performing algorithm is EGT-PC, using on average 76.57 μJ. This is because the average amount of time it requires to classify a packet is close to the maximum amount of time taken due to the configuration of its search structure. TSS is the second worst performing algorithm, using on average 53.25 μJ to classify a packet. It performed poorly due to the large amount of processing required, with each packet needing to perform direct memory lookups to convert its port numbers, the search of two LPM tries and the creation of the hash key required for each hash table lookup. HiCuts and HyperCuts showed similar performance, with HiCuts using on average 10.89 μJ to classify a packet and HyperCuts using 19.2 μJ. HyperCuts uses more energy on average classifying a packet when using the ruleset with 1000 rules than it does when using the ruleset with 1600 rules. This is due to the same reason that causes the dip in the number of worst case memory accesses. All five algorithms scaled well across the full range of rulesets tested.

Throughput

Fig. 2.11 shows the throughput for the five algorithms, and it can be seen that this is proportional to the average amount of energy used when processing a packet. This is good news as it means that algorithms with faster classification rates will have lower energy usage when operating on RISC type processors. The algorithm with the highest throughput is RFC, classifying on average 400,937 packets per second (p/s). This is followed by HiCuts, classifying on average 57,042 p/s, HyperCuts 32,242 p/s, TSS 10,700 p/s and EGT-PC 7,491 p/s.

2.3.4 Conclusions

The results presented give as fair a comparison as is possible of the five packet classification algorithms tested. They show that the algorithm with the smallest memory usage is HyperCuts. It only requires 56 KB of memory to store its search structure for the ruleset containing 2,191 rules. This is impressive when compared to the algorithm that uses the largest amount of memory, RFC. It requires over 3 MB of memory to store the search structure it built for the same ruleset. HyperCuts is also the best performing algorithm in terms of the amount of energy used building its search structure. The algorithm that performs best in terms of highest throughput and lowest amount of energy needed to classify a packet is RFC. This is because it requires the fewest number of memory accesses to classify a packet and it has the simplest search algorithm. HiCuts and HyperCuts came second and third respectively when it comes to highest throughput and lowest amount of energy needed to classify a packet.

It was with these points in mind that it was decided that HyperCuts would be the algorithm best suited towards hardware acceleration. The main reason for this is that its low memory usage allows it to build search structures for rulesets containing tens of thousands of rules that are small enough to fit in the on-chip memory of devices such as FPGAs, allowing for increased throughput. Its search algorithm is also suitable for hardware acceleration as it requires a small number of memory accesses and calculations to be performed when classifying a packet.

2.4 Deep Packet Inspection Systems

There are a wide range of network intrusion detection/prevention systems requiring DPI with Snort [23], Bro [40] and Cisco [41] being some of the most popular. Another popular system that employs DPI is Linux L7-filter [42] used to perform protocol analysis, categorising packets based on their payload content. The Linux and Cisco systems are signature-based, meaning that they only inspect a packet’s payload, while the Snort and Bro systems inspect both a packet’s header and payload. Regular expression matching is used to search for strings in the Bro, Linux and Cisco DPI systems, with Snort mainly using fixed string matching and more recently some regular expression matching. Snort, Bro and Linux are open source, with Snort being the most popular system, with millions of

downloads and over 250,000 registered users [43]. The Snort ruleset is also used as a testing benchmark for much of the prior art in the area of DPI. For these reasons the Snort 2.6.0 ruleset has been used to test the new string matching algorithm and hardware accelerator presented in this thesis.

The complexity of DPI systems means that they need to be implemented in software, limiting their packet processing throughputs to Megabits per second (Mbps) rather than Gbps, even when implemented on high-end processing systems. A performance evaluation [44] showed that the maximum throughput for Snort is around 51 Mbps when run using a Linux operating system and 82 Mbps when Windows Server 2003 is used. This is under normal traffic conditions using a Pentium 4 processor running at 3.2 GHz with 512 MB of Random Access Memory (RAM). The maximum throughput drops considerably when Snort is used to inspect malicious traffic, with 6 Mbps speeds recorded using Linux and 2.5 Mbps using Windows Server 2003. The following section explains Snort.

2.4.1 Snort

The Snort network intrusion detection/prevention system [23] is used to perform real-time traffic analysis and packet logging on IP networks. It can also perform protocol analysis, content searching/matching and can be used to detect a variety of attacks such as worms, viruses, Denial of Service attacks and other harmful activities. The Snort system is single threaded and consists of three main stages that process packets sequentially. The first stage uses a packet decoder to strip a packet of its Data Link Layer information. This information includes the packet's Ethernet header. The stripped packet is then passed to the next stage, where a pre-processor performs IP fragment and TCP stream reassembly. This data is then passed to a detection engine where most of the packet processing is performed. The detection engine is used to perform DPI, comparing packet header and payload information to thousands of rules. This engine can be configured to perform different actions depending on the rule matched or even if no rule is matched. These actions could be to allow the packet to pass through, drop the packet, log the packet or alert the administrator. Alerting the administrator of unusual activity would be an example of intrusion detection, while the dropping of a packet would be an example of intrusion prevention.

Each rule contains a *header rule* and a *content rule* as mentioned in Section 1.3.2. These rules are generated manually by skilled experts. They build rules based on known information. These rules are built by extracting unusual content from a packet's payload and header. As the number of known attacks and packets to be classified increases, so does the Snort ruleset. These rules contain thousands of unique strings that must be searched for in a packet's payload, but only a few hundred unique headers. This is because many rules will have a common *header rule*. The detection engine used in the Snort system matches rules using a rule chain logic structure. It works by first checking if the packet header matches any of the unique *header rules*. The more complex task of searching for a rule's *content rule* will be carried out for all rules that have had their *header rule* matched.

The matching of a *header rule* is an easier task compared to the matching of a *content rule* for a number of reasons. The first reason is that there are fewer unique *header rules* than there are unique *content rules* that need to be search for. The second reason is that the same fields are always used in the packet header to match the *header rule*, with these fields in a fixed location. These fields are the same as the fields used in single-match packet classification, including the source and destination IP address, the source and destination port numbers and the protocol number. The *content rule*, however, will contain strings of arbitrary length, with their starting location not always known, meaning that the entire packet payload may need to be searched.

2.4.2 Current Fixed String Matching Approaches

The area of fixed string matching is one of the best studied fields due to its many applications such as bibliographic search, word processing and use in Internet search engines. In recent times research has concentrated on its use in the area of DPI for intrusion detection/prevention systems. Some of the first and best known algorithms in the area of fixed string matching include the Knuth-Morris-Pratt [45] and Boyer-Moore [46] methods, which work well for single string matching. The performance of these algorithms actually improves if the length of the string being searched for increases. This is because they look at a window of characters in the text being searched equal in length to the string being sought. The

characters in the string being searched for are compared to the characters in the window of text being looked at, with failures at certain points allowing for the possibility that the window of text being looked at can move forward multiple characters at a time. This allows for a high average throughput, albeit with a poor worst case throughput of one character per cycle.

Algorithms that work well for matching multiple strings simultaneously include Aho-Corasick [37] and Commentz-Walter [47]. They do this through the use of a state machine built from the strings being searched for. The text being searched traverses this state machine from a root state at best one character at a time, using transition pointers stored at each state. The amount of memory needed to store the states and their transition pointers is a problem for these algorithms as their state machine memory footprint can grow exponentially in size as new strings are added. There has also been a host of other string matching algorithms and hardware accelerators offering improvements that seek to reduce memory consumption and increase throughput [48, 49, 50, 51, 52, 53].

Two algorithms are presented in [54] based on the Aho-Corasick approach to string matching. They are designed with hardware acceleration in mind and reduce memory consumption through the use of bitmaps and path compression. Path compression combines a series of successive states, each of which contain only a single pointer. This is done in order to reduce the total number of states that need to be stored. Bitmaps are used to reduce the number of pointers at a state from its worst case of 256. A problem with the use of bitmaps is the large logic delay required to find a pointer, slowing down the performance of any hardware implementation. Finding a pointer involves the checking and addition of the 256 bits contained within the bitmap, causing large logic delays. Both schemes also use fail pointers, meaning that they cannot guarantee the processing of a character on every clock cycle.

Another algorithm based on Aho-Corasick is presented in [55]. It splits the Aho-Corasick state machine into eight separate state machines. Each state machine is searched in parallel using one of the eight bits from the input character, reducing the maximum possible number of transitions at each state from 256 to 2. The results from each state machine are combined through the ANDing of bitmaps in

order to figure out if a match has occurred. A drawback of this design is that each state machine can only be used to search for a small number of strings as each state needs to store a bitmap whose bits represent the strings being sought. This means that many small state machines will be needed to store a ruleset containing thousands of strings.

In [56] bloom filters are used to implement a fixed string matching hardware accelerator. This approach can search for thousands of strings with very low memory consumption. All strings of the same length are placed in a separate bloom filter, with all filters inspecting the packet in parallel. The number of bytes inspected in a packet in a single clock cycle is equal to the shortest string length being searched for. Drawbacks with this approach are that rulesets such as Snort contain strings with many lengths, meaning that it is not possible to implement a bloom filter for all string lengths. Also, because of their structure, bloom filters only return that there is a possible match, meaning that an analyser must be used on the packet to check if the match was correct or a false positive.

2.4.3 Conclusions

Network intrusion detection/prevention systems such as Snort require fixed string pattern matching algorithms that are capable of searching for thousands of strings simultaneously in a packet's payload. Algorithms such as Knuth-Morris-Pratt and Boyer-Moore are therefore not suitable as they are only good at searching for single strings. The Aho-Corasick and Commentz-Walter algorithms can search for multiple strings but require large amounts of memory to save their state machines. Modified versions of the Aho-Corasick algorithm reduce memory consumption but cannot guarantee a fixed throughput or can only search for a small number of strings, while the algorithm that uses bloom filters is not suitable for searching for the type of strings used by Snort. The fixed string pattern matching algorithm and hardware accelerator presented in this thesis implement a modified version of the Aho-Corasick algorithm that uses default transition pointers to reduce memory usage. They can also search for thousands of strings with a guaranteed throughput.

2.5 Hardware-Based Platforms

There are a wide range of technologies that can be used to implement hardware accelerators designed to carry out the tasks of packet classification and string

matching. Each technology has its own advantages such as high speed or low power consumption. It can also have its own disadvantages such as high cost or poor flexibility. This means that it is important to carefully consider which technology the design of any new hardware accelerator is aimed at. This section reviews the three technologies that are most commonly used to implement hardware accelerators, stating their advantages and disadvantages.

2.5.1 ASIC

The use of Application Specific Integrated Chips (ASIC) for the implementation of hardware accelerators has many advantages and disadvantages. Advantages include the fact that a hardware accelerator implemented using an ASIC can have the highest throughput, lowest power consumption and smallest footprint of any hardware platform available. This is because the designer has complete control over the placing and routing of the logic and memory resources needed to implement a hardware accelerator. This complete control means that the delay path between logic components such as AND, OR and XOR gates can be kept as short as possible, allowing for the highest possible throughput. The designer can even have control of the process technology used to build the ASIC along with the type of transistors used to create the logic and memory elements. A hardware accelerator implemented using ASIC technology can be designed to have no surplus logic elements, helping to keep power consumption to a minimum.

Disadvantages with the use of ASIC technology are the long time and large financial cost in developing a hardware accelerator. This is due to the expense of licensing the logic and memory libraries along with the design software needed to design an ASIC, high manufacturing cost and the skilled design expertise required. The development of a hardware accelerator designed as an ASIC is very slow due to the large amount of testing a design must undergo before being put into manufacture. An ASIC also offers poor flexibility as it will only ever be able to implement the tasks that it was designed for.

2.5.2 FPGA

Field Programmable Gate Arrays (FPGA) offer an extremely flexible architecture for the implementation of energy efficient high throughput hardware accelerators.

They can run at speeds of a few hundred MHz, enabling the processing of network traffic at line speeds in excess of 40 Gbps, with substantially lower development and time to market costs than an ASIC. An FPGA contains programmable memory, logic and interconnect that can be configured to meet the designer's specific requirements. They also allow a wide range of external memory types such as Dynamic Random Access Memory (DRAM) and Static Random Access Memory (SRAM) to be used, increasing flexibility. The resources of an FPGA can be broken up into many different sub-blocks, with these blocks used to process data in parallel. This makes an FPGA ideally suited to the implementation of hardware accelerators for packet classification and string matching as multiple packets can be processed in parallel, allowing for large throughputs.

The Parallel String Matcher [57] by Titan-IC is a commercial hardware accelerator that can be implemented either on an FPGA or as an ASIC. It can be used to perform pattern matching for DPI, flow classification, TCP/IP header lookup, address translation, content/URL inspection/filtering and CAM emulation. It is able to perform 5-tuple packet classification for rulesets containing between 5-50 thousand rules, or string matching for rulesets containing between 1-10 thousand variable length strings. These tasks can be performed at speeds of between 120-200 Mpps when implemented using the internal memory of an FPGA built on 65nm process technology.

Another advantage that can be gained by the use of FPGAs is that it is a well developed technology, with companies such as Xilinx [58] and Altera [59] spending millions of dollars each year on research and development. This means that existing designs for hardware accelerators will be able to gain an increase in throughput and energy efficiency simply by porting to more modern FPGAs. A drawback that comes with using FPGA rather than ASIC technology is the increased power consumption due to the unneeded circuitry contained within an FPGA. Another drawback is reduced throughput due to the increased length of the interconnect used to join logic and memory elements.

2.5.3 TCAM

One of the most popular technologies for implementing packet classification hardware accelerators at present is Ternary Content Addressable Memory

(TCAM). TCAM is popular because it can match all rules from a ruleset in an O(1) clock cycle. This is achieved by carrying out parallel comparisons on all stored rules in a single clock cycle plus the use of pipelining. State of the art technology such as the Cypress Ayama 10000 Network Search Engine [20] can perform 133 million 144-bit search key per second. This high lookup rate, however, comes at a large cost of consuming between 4.86-19.14 Watts, depending on the TCAM size.

Besides the high power consumption, another drawback for TCAM is its poor storage efficiency of rulesets when using rules containing ranges. This is because a memory word's bits are stored in a 1, 0 or do not care state. This makes TCAM very efficient at storing fields that use longest prefix matching but poor at storing fields that use range matching. Range splitting must be performed to convert ranges into prefix formats. This further complicates the problem of power consumption as large amounts of memory are needed to store rulesets. Research of real world databases in [60] showed that TCAM storage efficiency ranged between 16-53%, with an average of 34%.

TCAMs also take up large amounts of die area, with one bit requiring 10-12 transistors, compared to SRAM, which only requires 4-6 transistors per bit and DRAM, which requires only 1 transistor and a capacitor. A search engine implemented using this approach will require multiple chips, including a host ASIC or FPGA, TCAMs and the corresponding SRAMs. Another problem with TCAM is its high price per bit due to the fact that it is a speciality type of memory and is not as commonly used as other memory types such as SRAM or DRAM. There has been much research [60, 61, 62] into reducing the power consumption of TCAM and increasing the storage efficiency of rulesets, but these issues still, however, remain a problem.

The use of TCAM for fixed string matching is not so common due to the fact that commercial TCAM only returns a single match, which is not a good feature when all matching strings are required. The use of do not care bits means that there can be many matches to the TCAM entries. TCAM will therefore only return the matching TCAM entry with the highest index number. Another drawback is that there will be a lot of memory wastage if the width of a TCAM entry is configured

to accommodate that of the longest string. The Snort ruleset uses strings with a large range of lengths. A TCAM-based multi-pattern matching scheme is presented in [63] that attempts to tackle these issues. It handles the issue of memory wastage associated with searching for long strings by breaking them up before storing them in TCAM. It searches through the packet one byte at a time by looking at a set of strings equal to the TCAM width. It records all partial matches and their position to identify if a full match has taken place. They deal with issues such as optimum TCAM width and are able to search for correlated patterns and patterns with negations. There are also other methods [64, 65, 66] for implementing fixed string matching through the use of TCAM. A drawback with all of these approaches is the high power consumption associated with TCAM.

2.5.4 Conclusions

The hardware accelerators presented in this thesis avoid the use of TCAM in order to keep power consumption to a minimum. They are instead implemented using FPGAs and as an ASIC in some cases. ASICs and FPGAs allow the use of on-chip SRAM which keeps throughputs high as external memory accesses are not required. Keeping the logic and memory on a single chip also has the advantage of allowing for a one-chip solution which further reduces power consumption. The flexibility of ASICs and FPGAs also means that they can implement multiple packet processing engines. This further increases throughput as multiple packets can be processed in parallel.

2.6 Low Power Design

The main goal of this thesis is to design energy efficient hardware accelerators for packet classification and string matching. It is therefore essential that power consumption is taken into account at all steps of the design process when trying to achieve high throughput. This section outlines the main causes of power consumption in Complementary Metal–Oxide–Semiconductor (CMOS) digital circuitry. It also discusses common design techniques that can be used when designing the architecture of a hardware accelerator, such as parallel processing and pipelining. These design techniques can be used to reduce power consumption whilst still achieving high throughput.

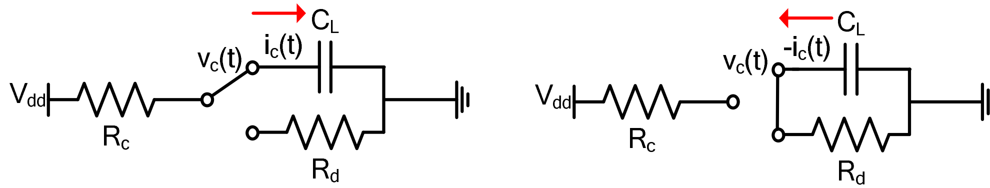


Fig. 2.12. Charging and discharging of a capacitive load.

2.6.1 Types of Power Dissipation

It is important to know the main types of power consumption in an integrated circuit and their causes before beginning the design of new hardware. Equation 2.1 shows the three main causes of power consumption in CMOS digital circuitry.

$$P_{Total} = P_{Dynamic} + P_{Short\text{-}Circuit} + P_{Static} \tag{2.1}$$

This includes $P_{Dynamic}$ and $P_{Short\text{-}Circuit}$, which are caused by switching and P_{Static}, which is a constant source of power consumption caused by current leakage. The rest of this section describes the cause of each of these types of power consumption in more detail.

Dynamic Power Consumption

The largest source of power consumption in a CMOS circuit is dynamic power, caused by the charging and discharging of a capacitive load [67, 68]. A CMOS inverter, which is made up of a PMOS and NMOS transistor, can be modelled using two resistors and a capacitor as shown in Fig. 2.12. The capacitance is present due to the unwanted parasitic effects between the tightly compacted wires and transistors that make up a circuit. The resistors R_c and R_d are the resistances of the charging and discharging circuits respectively. The switch is a model for the change in logic state, and the capacitor C_L is a model for the capacitive load. An input transition from one to zero will turn on the PMOS transistor, charging the capacitor. The resistance of the PMOS transistor is modelled by R_c, with the current i_c charging the capacitive load. The energy used as the capacitor is charged from time t_0 to t_1 can be calculated using Equation 2.2.

$$E_{charge} = \int_{t0}^{t1} V_{dd} i_c dt = C_L V_{dd} \int_{t0}^{t1} \frac{dv_C(t)}{dt} dt = C_L V_{dd}^{\;2} \tag{2.2}$$

Half of the energy is stored in the capacitor and the other half is dissipated as heat in the resistor R_c. The energy stored in the capacitor E_{cap} can be calculated using

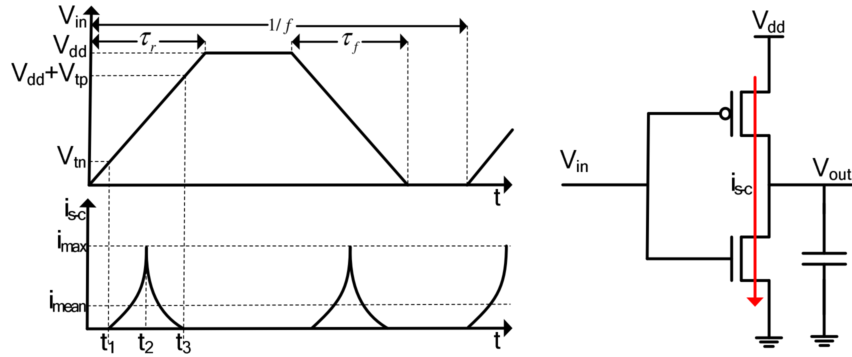


Fig. 2.13. Switching characteristics of a CMOS inverter.

Equation 2.3, while Equation 2.4 can be used to calculate the energy dissipated by the resistor E_c.

$$E_{cap} = \int_{t0}^{t1} V_c(t) i_c(t) dt = \frac{1}{2} C_L V_{dd}^{\ 2} \tag{2.3}$$

$$E_c = E_{charge} - E_{cap} = \frac{1}{2} C_L V_{dd}^{\ 2} \tag{2.4}$$

An input transition from zero to a one will turn on the transistor NMOS. This will discharge the capacitor through the NMOS transistor, whose resistance is modelled by R_d. The energy in the capacitor is dissipated as heat in the resistor R_d. The energy dissipated will be equal to E_{cap} if the capacitor is given time to fully discharge. The dynamic power consumption of a circuit can be calculated using Equation 2.5, where f is the clock frequency in Hz and α is the probability of C_L being charged or discharged.

$$P_{Dynamic} = C_L V_{dd}^{\ 2} f \alpha \tag{2.5}$$

Short-Circuit Power Consumption

Short-circuit power is a source of power consumption in CMOS circuitry that is caused by PMOS and NMOS transistors both being on at the same time during the switching of input signals. Fig. 2.13 is used to highlight this phenomenon, showing a CMOS inverter and its switching characteristics. Only one transistor should ever be on in normal operation. The input signals, however, have a finite rise and fall time, which means that both transistors will be on for a very short amount of time. The term for the dynamic power consumption derived in the last

section did not take these rise and fall times into account. Both transistors being on will cause a direct current path between the supply voltage and ground.

For simplicity it can be assumed that $\beta_p = \beta_n = \beta$ (where β is the gain of a transistor). It can also be assumed that $-V_{Tp}=V_{Tn}=V_T$ (where V_{Tp} is the threshold of the PMOS transistor and V_{Tn} is the threshold of the NMOS transistor) and that time period t_1-t_3 is symmetrical with respect to t_2. This leads to Equation 2.6 for the mean current over one time period [69].

$$I_{mean} = 2\frac{2}{1/f}\int_{t1}^{t2} I(t)dt = 4f\int_{t1}^{t2}\frac{\beta}{2}(V_{in}(t)-V_T)^2 dt \tag{2.6}$$

Equation 2.7 also gives the mean current, where $V_{in}(t) = V_{dd}t/\tau$, assuming that the input signal is symmetrical with equal rise and fall times ($\tau_r = \tau_f = \tau$) and that there is a linear relationship between the input voltage (V_{in}) and time (t) during transitions. In this equation t_1 is expressed as $(V_T.\tau)/V_{dd}$ and t_2 as $\tau/2$.

$$I_{mean} = 2\beta f\int_{\tau/2}^{(V_T.\tau)/V_{dd}}\left(\frac{V_{dd}.t}{\tau}-V_T\right)d\left(\frac{V_{dd}.t}{\tau}-V_T\right) \tag{2.7}$$

The solution for this is given in Equation 2.8.

$$I_{mean} = \frac{1}{12}\frac{\beta}{V_{dd}}(V_{dd}-2V_T)^3\tau f \tag{2.8}$$

The short-circuit power consumption can therefore be expressed using Equation 2.9 where $P_{Short\text{-}Circuit}=I_{mean}V_{dd}$.

$$P_{Short-Circuit} = \frac{\beta}{12}(V_{dd}-2V_T)^3\tau f \tag{2.9}$$

It can be seen that the short-circuit power consumption can be reduced by decreasing the rise and fall times of the input signals. This, however, would come at the expense of increased power consumption in the circuitry generating the input signals. Using a large capacitor would also decrease the short-circuit power consumption as the output voltage would respond more slowly, resulting in both transistors being on for a shorter amount of time. A larger capacitor would, however, increase the dynamic power consumption. This is not therefore worth doing as short-circuit power consumption is typically small, only consuming 10% of the power used by dynamic power consumption [70].

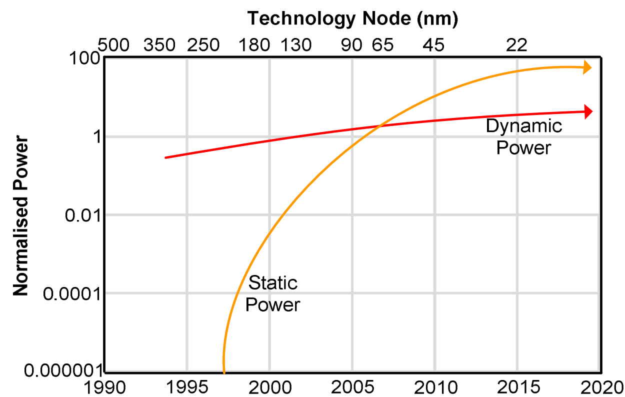


Fig. 2.14. Static vs. dynamic power.

Static Power Consumption

A CMOS circuit should ideally consume no power when it is in a steady state with no switching taking place. This, however, is not the case as there is a constant source of power consumption known as static power that is caused by sub-threshold current leakage and reverse biased diode junction current leakage. Dynamic power consumption has historically been the main cause of power consumption in a CMOS circuit, with static power consuming a much smaller percentage. The trend of implementing CMOS circuits using ever smaller process technologies has meant that static power is starting to use a much larger percentage of the power used. This is due to the fact that the dynamic power is proportional to the square of supply voltage, and supply voltage is reduced each time a smaller process technology is used. This means that reducing the supply voltage significantly reduces the dynamic power. The use of smaller process technologies worsens current leakage, meaning that it could become the main source of power consumption in the future. Fig. 2.14 shows a graph highlighting this trend [71].

The sub-threshold current leakage is caused by current flowing from a transistor's source to its drain, even if the gate to source voltage is lower than the transistor's threshold voltage V_T. This occurs because of carrier diffusion between the source and drain regions of the CMOS transistor in weak inversion. Sub-threshold current leakage will become significant when the gate to source voltage is just below the threshold voltage of the transistor. Equation 2.10 gives the formula for calculating the sub-threshold power consumption [72] where K and n are experimentally derived, W is the gate width, V_0 is the thermal voltage (about

25mv at room temperature), V_{dd} is the source supply voltage and V_j is the voltage across the junction.

$$P_{sub} = V_{dd} K W e^{-V_T / nV_0} (1 - e^{-V_j / V_0}) \tag{2.10}$$

The other source of static power consumption is reverse biased diode junction current leakage, caused by parasitic diodes that form between the diffusion region of a transistor and the substrate. It can be calculated using Equation 2.11 [70] where I_s is the reverse current in a diode caused by the diffusion of minority carriers from the neutral region to the depletion region.

$$P_{junc} = V_{dd} I_s (e^{V_j / V_0} - 1) \tag{2.11}$$

The work in [73] investigates various methods for reducing static power consumption such as turning off unused devices, using less leaky transistors and partitioning the design to allow for lower supply voltages. A dual threshold technique is introduced in [74] that assigns high thresholds to transistors in the non-critical path and low thresholds to transistors in the critical path. This allows transistors in the critical path to be fast but means that they consume a lot of static power, while the transistors in the non-critical path are slow but consume very little static power.

2.6.2 Power Benchmarking

The power consumption of the logic used in the ASIC implementation of the low power packet classifier presented in Chapter 4 has been estimated using a Taiwan Semiconductor Manufacturing Company (TSMC) 65nm low power process technology. Due to licensing issues the power consumption of the memory used in this ASIC implementation has been estimated using Chartered Semiconductor Manufacturing 130nm dual and single port RAM compilers. A method for normalising the power consumed is therefore needed so that the power consumed by circuits implemented using different process technologies that operate at different voltages can be compared. The normalisations used in this thesis ignore leakage power and assume that dynamic power is the major component. This assumption gives good first order normalisations [75] and is true for the libraries used in the ASIC implementations here, with leakage power being two orders of magnitude less than the dynamic power consumption [76]. The equation for

dynamic power consumption (Equation 2.5) is restated here for convenience as Equation 2.12.

$$P = C_L V^2 f\alpha \tag{2.12}$$

The load capacitance of a transistor C_L can be expressed using Equation 2.13 [68, 75] (This is the gate capacitance of the transistor and ignores other gate and interconnect parasitics, which scale similarly). The permittivity of the gate oxide is represented by ε_0 in this equation, L is the channel length, W is the channel width and H is the gate oxide thickness.

$$C_L = \frac{\varepsilon_0 \times L \times W}{H} \tag{2.13}$$

The frequency f and switching probability α do not have to be scaled as they are independent of the process technology used. The channel length and width of the transistor are scaled by a factor S, while the gate oxide thickness and voltage are scaled by a factor U. This leads to Equation 2.14, which can be used to normalise P with respect to V and Equation 2.15, which can be used to scale C_L.

$$P'(V) = P \times U^2 \tag{2.14}$$

$$C_L' = \frac{\varepsilon_0 \times (L \times S) \times (W \times S)}{(H \times U)} = \frac{\varepsilon_0 \times L \times W}{H} \times \frac{S^2}{U} = C_L \times \frac{S^2}{U} \tag{2.15}$$

Equation 2.16 can therefore be used to normalise P with respect to V and L.

$$P'(V, L) = P \times U^2 \times \frac{S^2}{U} = P \times S^2 \times U \tag{2.16}$$

2.6.3 Low Power Design Techniques

It is widely recognised that power consumption should be factored into the design of new hardware accelerators at all stages [77, 78, 79], especially at the higher levels of the design stage, as this is where the most design freedom exists and is where the most power can be saved. It is estimated that power savings of up to 20× can be made at the system design stage, compared to savings of less than 20% at the design layout stage [78].

Algorithmic

Large savings in power consumption can be made by keeping the amount of tasks an algorithm has to perform when processing data to a minimum. Reducing the

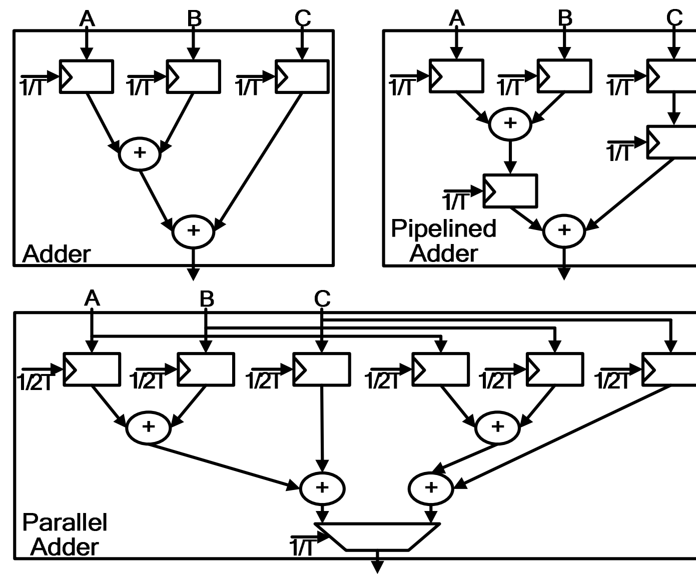


Fig. 2.15. Implementation of a parallel and pipelined three input adder.

number of tasks that need to be performed will reduce the amount of switching and time taken when processing data. A reduction in the amount of switching will lower the dynamic and short-circuit power consumption, whilst a reduction in processing time will allow clock speeds and voltage levels to be reduced, with the same level of throughput maintained. Efforts should also be made to keep the processing tasks as simple as possible so that the amount of hardware required is kept to a minimum. This will reduce the amount of transistors required to implement a design, reducing the amount of static power consumed because of leakage current. The packet classification and string matching algorithms presented in this thesis have been carefully designed so that the hardware accelerators implementing them do not need to perform any logic intensive tasks such as floating point division. Their design means that only simple tasks such as shifting and addition need to be performed when processing data.

Architectural

There is also scope for large power savings at an architectural level, after the algorithmic details have been decided on. Techniques that can be used at an architectural level to reduce power consumption include parallel processing and pipelining. These techniques allow a targeted level of throughput to be reached with reduced clock frequencies and voltage levels. Fig. 2.15 shows an example where a simple three input adder has been implemented using parallel processing

and pipelining. Parallel processing can be implemented if the amount of area available to lay out a design is not tight. It involves increasing the amount of processing modules available to carry out certain computational tasks. This allows clock frequencies and voltage levels to be reduced as more modules are available to process data. The disadvantage of parallel processing is that extra resources are required to implement a design. Pipelining involves breaking up a design into stages, with each stage separated by registers. Breaking a design up into stages will reduce the length of the critical path, allowing the same clock frequency to be obtained at a reduced voltage level. The disadvantage of pipelining is that it will add extra delay to the amount of time that it takes to process data.

An analysis of the power reduction that can be achieved by implementing parallel processing and pipelining was carried out in [80] on a simple design consisting of an adder and a comparator. It found that power consumption could be reduced by 64% if parallel processing was carried out, with the computational resources doubled. This increased the board area by a factor of 2.15, with the clock frequency and voltage levels reduced by 50% and 42% respectively. It also found that power savings of 61% could be made by using pipelining, with the board area increased by a factor of 1.15 and the voltage levels reduced by 42%. Power savings of 80% were made by implementing a combination of parallel processing and pipelining.

Register Transfer Level (RTL) Coding

The power savings that can be made by carefully coding a design using a RTL Hardware Description Language (HDL) such as VHDL or Verilog are significantly less than the savings that can be made at an algorithmic or architectural level. However, they are still worth considering as even a power saving of only a few percent can be important if power budgets are tight. Simple coding techniques that can be used to reduce power consumption include using one-hot or grey coding in state machines to reduce the amount of switching activity. Switching can also be reduced by enabling all registers so that data only changes on their output when required. Another method for saving power is to balance the logic within data paths so that data arrives to the input of logic modules at the same time. This minimises the glitching that occurs as signals settle to their final values.

Implementation and Layout

Great care should also be taken when laying out a design as an ASIC or on the chosen FPGA so that power consumption is kept to a minimum. A design should be laid out so that the paths that have the heaviest switching load are kept as short as possible. Careful consideration should also be given to the amount of input/output pins used and their positioning, as it is estimated that they can cause 33% of the total power consumption [79]. Using an ASIC with on-chip memory or the internal block RAM of an FPGA where possible will lead to large power savings as the routing interconnect and number of input/output pins can be greatly reduced. The algorithms presented in this thesis go to great effort to keep memory usage as small as possible, so that the hardware accelerators that implement them can use on-chip memory, keeping power consumption to a minimum.

2.7 Summary

This chapter has provided detailed background information into the areas covered in the remainder of this thesis. It has explained the structure of the rulesets used for testing the packet classification algorithm and hardware accelerators that are presented in Chapters 3 and 4. A detailed survey of the most commonly used software approaches for implementing packet classification was also carried out, identifying the HyperCuts algorithms as being an ideal contender for hardware acceleration. This is because it scales well in terms of memory consumption and throughput when large rulesets are used. An explanation of network intrusion detection/prevention systems was given next, with particular attention given to Snort, as it is the system that relies most heavily on fixed string matching when detecting intrusions. Snort is also given particular attention as its ruleset is used to test the new fixed string matching algorithm and hardware accelerator presented in Chapter 5. A detailed survey was carried out on approaches used for fixed string matching to give context to the work presented in Chapter 5. Popular hardware platforms for implementing such hardware accelerators and their sources of power consumption were also given, along with design methods that should be used when implementing an energy efficient hardware accelerator.

Chapter 3 - Packet Classification Architectures

3.1 Introduction

Packet classification is used by networking devices to carry out advanced Internet services like network security, sophisticated traffic billing, giving priority to VoIP and IPTV packets, rate limiting, load balancing, NAT and resource reservation. It is a complex task that needs to be carried out using devices such as programmable multi-core network processors. The flexibility of these devices reduces their throughput, limiting packet classification to edge routers where line speeds are typically only a few Gigabits per second. Analysis of popular packet classification algorithms in Section 2.3.3 showed that even the best performing algorithm in terms of throughput RFC [12] can only classify around 400,000 packets per second. This is when it is implemented in software and run on an SA1100-StrongARM RISC processor similar to the type used as the processing cores in many of today's programmable network processors. Current commercial hardware approaches that could allow packet classification to be performed at core network line speeds of up to 40 Gbps use large amounts of power. The Cypress Ayama 10000 Network Search Engine [20], for example, uses up to 19.14 Watts when classifying 125 million packets per second. The structure of TCAM also makes it poor at storing large rulesets due to its difficulty in storing rules that contain ranges.

This chapter introduces novel hardware architectures for packet classifiers that can be implemented using an FPGA or as an ASIC. They are capable of handling line speeds in excess of 40 Gbps for rulesets containing tens of thousands of rules, allowing packet classification to be performed at core network line speeds. The architectures use energy efficient memories that are well suited to storing packet

classification rulesets. A modified version of the HyperCuts [10] packet classification algorithm is used to build the search structures for these architectures. The architectures are divided into two types, with one type using ultra-wide memory words and the other using reduced width memory words. The hardware accelerator that uses ultra-wide memory words performs well when using rulesets that contain a lot of wildcard rules, while the hardware accelerator that uses reduced width memory words can achieve higher throughput and performs well when using rulesets that do not contain a lot of wildcard rules.

The hardware accelerator architectures presented in this chapter implement modified versions of the HyperCuts packet classification algorithm. Section 3.2 therefore gives a detailed explanation of the HyperCuts algorithm, which was briefly explained in Section 2.3.1. This is done so that the modifications made to make the algorithm more suited to hardware acceleration can be better understood. These modifications are explained in Section 3.3. The architecture of the logic used to select the correct path as a packet traverses the decision tree is common to all architectures presented and it is explained in Section 3.4. The memory organisation of the search structures built for the different hardware accelerator architectures are explained in Section 3.5. Section 3.6 explains the architecture of the different packet classification engines used, while Section 3.7 explains how they can be configured to work in parallel. The performance results including memory usage, throughput, and power consumption are presented in Section 3.8. This section also compares the performance of the hardware accelerators against prior art. Section 3.9 concludes the chapter.

3.2 Decision Tree-Based Packet Classification

The linear search of a packet's header against each rule in a ruleset for a match will result in an unacceptably large worst case amount of processing time, preventing a classifier from classifying packets at the speeds required for it to work at the core or even edge of a network. This worst case amount of processing time can be reduced by using the HyperCuts packet classification algorithm. It is a decision tree-based algorithm that builds a search structure that allows incremental updates to a ruleset. Search structures that allow incremental updates do not have to be rebuilt each time a ruleset has a rule added or deleted.

HyperCuts works by breaking a ruleset into different groups, with each group containing a small number of rules suitable for a linear search. The maximum number of rules that can be contained within a group is limited using a predefined number known as *binth* to ensure that only a short linear search is required. Each group of rules is stored in a leaf node of a decision tree, with a packet finding the leaf node that contains the matching rule by traversing the decision tree using values from its header to guide it.

HyperCuts creates this decision tree by taking a geometric view of a ruleset, with each rule considered to be a hypercube in hyperspace. The boundaries of each hypercube are defined by the range specifications of the rule it represents. The algorithm cuts into this hyperspace by performing cuts to the fields used to define it. Each cut will create sub-regions, with each sub-region containing the rules whose hypercubes overlap. The information regarding the first set of cuts used to divide the hyperspace is stored in the root node of a decision tree. This information includes the number of cuts that are to be performed to each field and the memory location of each of the resulting sub-regions. These sub-regions are known as the root's child nodes, with sub-regions that contain no rules known as empty nodes. Sub-regions whose number of rules does not exceed the *binth* value are known as leaf nodes. Each leaf node stores one rule group that can be searched linearly. A sub-region that contains more rules than is allowed by the *binth* value is known as an internal node and the space it occupies must be further broken up into smaller sub-regions. This internal node will store information specifying the number of cuts that must be performed to each field used to split the space it occupies into smaller sub-regions. It also stores the memory location of the resulting sub-regions that are the internal node's child nodes. An internal node can also have empty, leaf and internal nodes. The dividing of the hyperspace into ever-smaller sub-regions will end when the number of rules in all sub-regions does not exceed the *binth* value.

The algorithm uses a set of rules to determine the fields that should be considered for cutting the hyperspace covered by an internal or root node. It examines the rules that overlap the hyperspace being cut, calculating the number of distinct range specifications for each field. It then selects the fields for cutting whose distinct number of range specifications is greater than or equal to the mean number

Table 3.1. Sample ruleset containing nine rules.

RuleID	S. IP	D. IP	S. Port	D. Port	Protocol	Action
R_1	111*	010*	78-78	230-702	UDP	ACT_1
R_2	111*	1***	0-2000	10-10	UDP	ACT_2
R_3	1***	101*	60-80	0-65535	TCP	ACT_3
R_4	10**	000*	0-65535	960-990	TCP	ACT_4
R_5	00**	101*	0-65535	800-811	TCP	ACT_5
R_6	000*	0111	30-80	0-65535	UDP	ACT_6
R_7	00**	0101	30-80	0-65535	TCP	ACT_7
R_8	000*	0100	30-80	0-65535	UDP	ACT_8
R_9	001*	0110	0-65535	0-65535	UDP	ACT_9

of distinct range specifications. HyperCuts also has a rule for limiting the number of cuts that the combination of cuts between the chosen dimensions can equate to in order to prevent the decision tree from using up large amounts of memory. The maximum number of cuts that can be made to an internal or root node is specified by Equation 3.1.

$$\textit{max cuts to node } i \leq \textit{spfac*sqrt(number of rules at i)} \qquad (3.1)$$

Where *i* is the internal or root node being cut and *spfac* is a user defined value used to control memory usage. Small *spfac* values will result in fewer cuts to nodes, creating a deep and narrow decision tree, while large values for *spfac* will allow more cuts, resulting in a wide but shallow decision tree. A deep and narrow decision tree will generally require less memory but will have a larger worst case processing time when matching a packet to a rule as more internal nodes will need to be traversed. The HyperCuts algorithm does not make it clear how to choose the best combination of cuts among the fields chosen to cut an internal or root node. Here all possible combination of cuts between the chosen dimensions are considered that conform to the equation limiting the maximum number of cuts that can be made to an internal or root node. The maximum number of rules stored in a child node for each combination of cuts is recorded, with the combination that results in the smallest number of maximum rules stored in a child node chosen.

3.2.1 Building a Decision Tree

This section describes step by step how to build a decision tree from the ruleset shown in Table 3.1. The source and destination IP addresses have been reduced from 32 to 4 bits to aid the explanation. The first step in building the decision tree

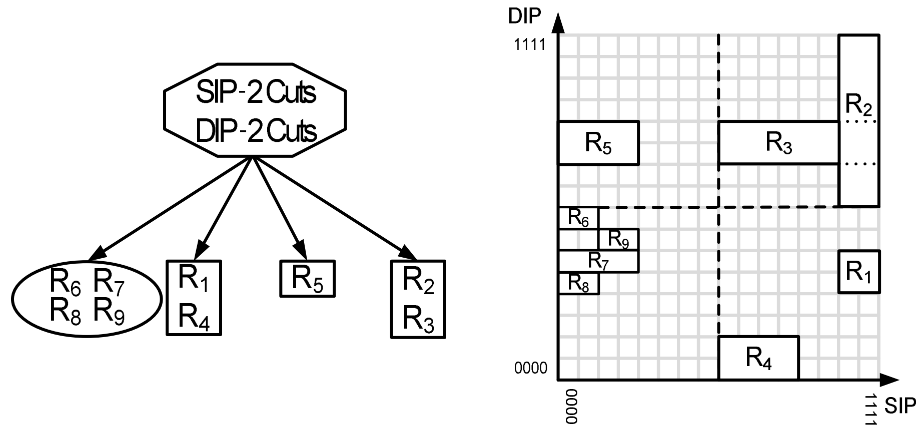


Fig. 3.1. Cuts performed to the root node of a decision tree.

is to decide a value for *spfac* and *binth*. In this example they will both be set equal to two. The next step involves deciding which dimensions should be used by the root node to cut the hyperspace. This is done by first calculating the number of distinct range specifications for each field, with the source IP address having six, the destination IP address having eight, the source and destination ports both having five and the protocol number having two, giving a mean number of 5.2. The source and destination IP addresses shall therefore be considered for cutting as they both have a distinct number of range specifications greater than the mean. The maximum number of cuts that can be performed to the root node is calculated next using Equation 3.1, limiting the maximum number of cuts to six. The number of cuts that can be performed to a node is limited to be a power of two for ease of implementation, which means a maximum of four cuts can be performed.

The next step involves trying all combinations of cuts between the chosen dimensions that are less than or equal to four, with the maximum number of rules stored in a child node for each combination of cuts recorded. The combinations of cuts that can be made to the source and destination IP address are [0, 2], [0, 4], [2, 0], [2, 2] and [4, 0]. The combination that results in the smallest maximum number of rules stored in a child node is to cut both the source and destination IP address in two. Fig. 3.1 shows how the decision tree will look after performing these cuts. It also includes a geometric representation of the source and destination IP addresses, showing the cuts made to the root node (represented by an octagon in the decision tree). It can be seen that these cuts create four sub-regions. Three of these sub-regions conform to the *binth* value as they contain two

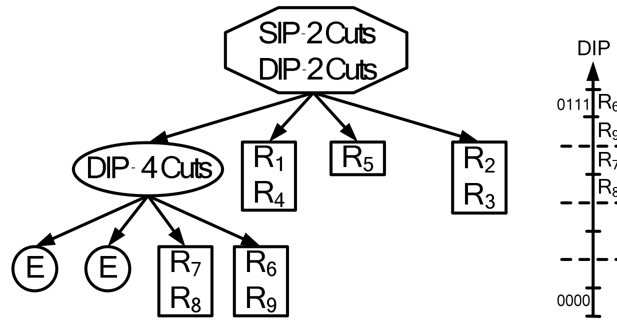


Fig. 3.2. Cuts performed to the internal node of a decision tree.

or less rules. This means that they are leaf nodes (represented by rectangles in the decision tree). The fourth sub-region contains more rules than the *binth* value allows. This means that it is an internal node (represented by an oval in the decision tree) that must be cut further.

The first step that must be carried out when cutting the internal node is to decide which dimensions should be considered for cutting. This is done by calculating the number of distinct range specifications for each field using the rules contained within the sub-region. This time the source IP address has three distinct range specifications, the destination IP address has four, the source port and protocol number has two and the destination port has one, giving a mean number of 2.4. The source and destination IP addresses are again considered for cutting as they both have a distinct number of range specifications greater than the mean. Equation 3.1 is used again to calculate the maximum number of cuts that can be performed to the internal node, which is four in this case. The combinations of cuts that can be made to the source and destination IP address are the same as the combinations tried when cutting the root node. This time the combination that results in the smallest maximum number of rules stored in a child node is to perform four cuts to the destination IP address. This results in four sub-regions, with all sub-regions containing two or less rules, which means that they all conform to the *binth* value and no more cutting needs to take place.

Fig. 3.2 shows the finished decision tree and the cuts performed to the destination IP address when cutting the internal node. It can be seen that two of the sub-regions contain no rules which means that they are empty nodes (represented by circles in the decision tree). The remaining two sub-regions are stored as leaf nodes. A packet with a header value [0001, 0111, 50, 80, UDP] would traverse the

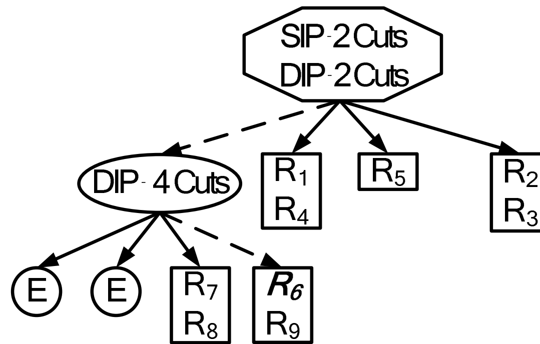


Fig. 3.3. Traversing a decision tree to find a matching rule.

decision tree to find a matching rule in the following manner, with Fig. 3.3 showing the path traversed. The root node is first looked at and it can be seen that it specifies that two cuts must be performed to both the source and destination IP address. This is done by examining the MSB of each header field. Only one bit needs to be examined for each field, as each field only has two cuts, which can be represented by one bit. The MSB for each field in this case is [0001, 0111]. These bits are concatenated to form the index 00, which represents the child node that must be traversed to. This child node is an internal node, meaning that more cuts need to be performed to the packet header in order to find the appropriate leaf node to search. The internal node is split by performing four cuts to the destination IP address. The next two MSBs must therefore be examined in the destination IP address of the packet header as two bits are needed to represent the four possible cuts. The value of these bits are [0111] giving the index 11, which represents the child to be traversed to. This child is a leaf node, which is searched linearly by comparing each of the rules to the packet header one by one until a match is found. This will return rule R_6 as the matching rule in this example.

3.2.2 Heuristics Used to Reduce Memory Usage

The HyperCuts packet classification algorithm uses different heuristics to minimise the amount of memory needed to save a decision tree and reduce the number of memory accesses required to match a rule. These heuristics are illustrated in Fig. 3.4.

- The first heuristic is called *Node Merging*, which is used to avoid the replicated storage of identical nodes. Node Merging is carried out by first

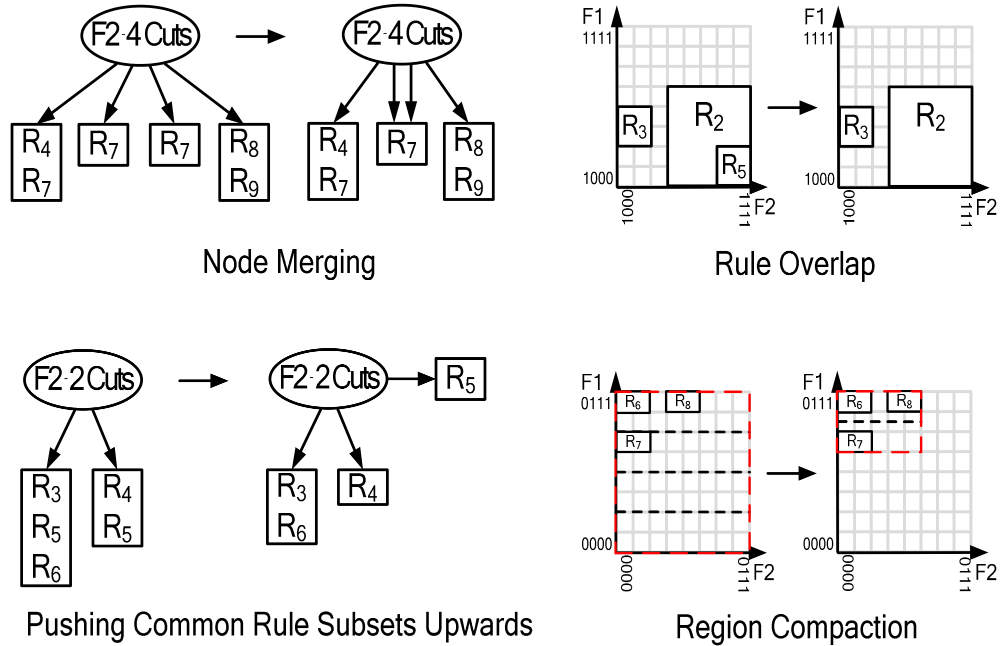


Fig. 3.4. Heuristics used by HyperCuts to reduce memory consumption.

searching the decision tree for leaf nodes that contain the same list of rules. The pointers to these nodes (stored in root and internal nodes) are then modified so that they point to just one of these leaf nodes, meaning that multiple copies do not need to be stored.

- HyperCuts uses a second heuristic called *Rule Overlap* to avoid the storage of rules in leaf nodes that can never be matched. A rule can never be matched and is therefore removed from a leaf node if the hypercube of a rule with a higher priority completely covers the space it occupies within the leaf node's sub-region.

- A third heuristic used to avoid the replicated storage of rules is called *Pushing Common Rule Subsets Upwards*. This heuristic stores rules at a parent node that would otherwise need to be stored in all its child nodes. Internal and root nodes could also need to be searched if this heuristic is used.

- The final heuristic used is called *Region Compaction* and it is employed to aid in the more efficient cutting of the hyperspace. Each node in a decision tree will cover a specific region of the hyperspace. The rules associated with a node may, however, cover a smaller region. Region Compaction shrinks the

area covered by a node so that it only covers the minimum amount of hyperspace that will cover all rules associated with the node. This means that a smaller region will need to be cut when dividing the hyperspace occupied by a node into sub-regions. This could result in fewer cuts, reducing memory consumption.

3.3 Algorithmic Modifications

The HyperCuts algorithm works well when implemented in software and run on a general purpose processor. It is not, however, optimised for implementation using dedicated hardware. This section explains the modifications made to the cutting scheme, region compaction heuristic and rule storage method in order to make the algorithm better suited to hardware acceleration. The modified cutting scheme improves throughput by making the decision tree as shallow as possible so as to reduce the number of memory accesses required to classify a packet. It can easily be configured to build search structures tailored to architectures with different width memory words.

The region compaction scheme introduced in the HyperCuts algorithm is modified because it requires floating point division to be carried out when a packet traverses the decision tree. It also requires the minimum and maximum values of the area covered by all fields to be stored at a decision tree's internal and root nodes so that it is possible to calculate the child node to be traversed to. An alternative scheme is introduced here that uses pre-cutting to compact the region covered by a node more intelligently so that floating point division does not need to be performed when traversing the decision tree. The new scheme instead uses only simple shift and AND operations when deciding which path to take when traversing the decision tree. Using pre-cutting to compact the region to be cut also has the advantage of not requiring the minimum and maximum values for each field to be stored at an internal or root node, reducing memory consumption. The removal of floating point division simplifies the hardware accelerator's architecture, allowing for increased speed and reduced power consumption. Pre-cutting is explained in detail in Section 3.3.2.

The method for storing rules in a leaf node is also modified here by using simple compression techniques to lower memory consumption and reduce the required

number of memory accesses needed to search a leaf node. The pushing common rule subsets upwards heuristic is not used as it was found during testing of ACL, FW and IPC rulesets to make only a fractional reduction in memory usage. It would also result in a more complicated search structure that would slow down the hardware accelerator as it would have to be able to search root, internal and leaf nodes for matching rules. Pushing common rule subsets upwards can also add extra memory accesses when classifying a packet. This is because a leaf node might still need to be searched even if a matching rule is found at an internal or root node. This is due to the fact that a leaf node might contain an alternative matching rule with a higher priority. Such a case would mean that the search of the rules at internal or root nodes was needless. Another disadvantage with this heuristic is that the number of rules stored at a parent node could exceed the limit on the maximum number of rules that can be stored in a leaf node. This would lead to excessively long search times.

3.3.1 Cutting Scheme

The cutting scheme employed to build the search structures used by the hardware accelerator architectures requires three pieces of information to be specified before building of the decision tree can begin. This information includes:

- The number of cuts to be performed to the root node.
- The maximum number of cuts that can be performed to an internal node.
- The maximum number of rules that can be stored in a leaf node.

The cutting scheme performs the majority of cuts to the root node because this will result in a shallow decision tree with the leaf nodes located closer to the root of the decision tree. The number of cuts that can be performed to an internal node is limited to only a few cuts to prevent the decision tree from using too much memory. It also means that the information needed to traverse an internal node can be placed in a single memory word, allowing them to be traversed in a single clock cycle. The hardware accelerator designed to use ultra-wide memory words can hold 48 rules on each memory word, which can be accessed and searched in a single clock cycle. It therefore limits the number of rules that can be stored in a leaf node to multiples of 48. Such large leaf nodes mean that only a small number

of cuts are required to divide the hyperspace into sub-regions whose number of rules do not exceed the maximum limit. The hardware accelerators designed to use reduced width memory words limit the number of rules that can be stored in a leaf node to multiples of two as they can only store two rules on each memory word. These architectures therefore need to perform a large number of cuts to the hyperspace so that the resulting sub-regions do not exceed the maximum limit on the number of rules that they can contain.

The algorithm begins by first performing the required number of cuts to the root node. The number of cuts must be 2^n where n can be any whole number in the range 1-9 if the architecture that uses ultra-wide memory words is used. A limit of 512 cuts is placed on the root node because the memory words have been designed so that they are wide enough to hold all the information required to store an internal or root node. The memory words are 7,704 bits wide, which leaves only enough room to store the root node's cutting information and pointers for 512 child nodes. The architectures that use reduced width memory words limit n to any whole number between 1-18 if the architecture that uses internal memory is used and 1-19 if the architecture that uses external memory is used. Caps of 262,144 and 524,288 cuts respectively are used because of limitations on the amount of memory available to save the search structures. These architectures require two memory accesses to traverse a root node, with one memory access used to retrieve the root node's cutting information and another to retrieve the memory address of the child node to be traversed to.

The algorithm uses the same method employed by HyperCuts to select the fields that should be considered for cutting. It only considers fields whose number of distinct range specifications is greater than or equal to the mean number for all fields. All combinations of cuts between the chosen fields that equal the 2^n limit are tried on the root node. The child node with the maximum number of stored rules is recorded for each combination of cuts, with the combination where this number is smallest chosen.

The algorithm searches through all child nodes created from cutting the root node, with more cuts performed to the nodes whose number of rules exceeds the maximum specified limit. The number of cuts that can be performed to the

Table 3.2. Maximum number of cuts allowed by the cutting scheme.

Architecture	Max Cuts to Root Node	Max Cuts to Internal Node	Width of a Memory Word
Ultra-wide memory	512	512	7,704-bit
Reduced width memory (internal)	262,144	16	324-bit
Reduced width memory (external)	524,288	4	288-bit

internal nodes is the same as the number that can be performed to the root node for the architecture that uses ultra-wide memory words. This is because it only allows a small number of cuts to both internal and root nodes, which it can traverse in a single clock cycle. The number of cuts that can be performed to the internal nodes for the architectures that use reduced width memory words is 2^m, where m can be any whole number between 1-4 if internal memory is used and 1-2 if external memory is used. The number of cuts that can be performed to an internal node has been capped at 16 and 4 respectively so that all the information needed to traverse an internal node can fit in a single memory word, allowing them to be traversed in a single clock cycle. Information on the number of cuts allowed to the root and internal nodes for the different architectures is shown in Table 3.2.

The architecture that uses internal memory can perform more cuts to the internal nodes because it uses wider memory words, allowing it to store more pointers. The architecture that uses external memory also has to store more information with each of its pointers, as explained in Section 3.5.2. Limiting the number of cuts also prevents excess memory usage and reduces the amount of time required to build the decision tree. The cutting of an internal node differs from the cutting of a root node in that all combinations of cuts are tried between the dimensions chosen for cutting that are less than or equal to the maximum limit. All combinations of cuts that are less than or equal to the maximum limit can be tried because there are only a few valid combinations that can be tried quickly. Cutting is complete when the number of rules in all sub-regions does not exceed the maximum specified limit.

3.3.2 Region Compaction

This section begins by giving a detailed explanation of the region compaction heuristic used by HyperCuts so that the modifications made here can be better

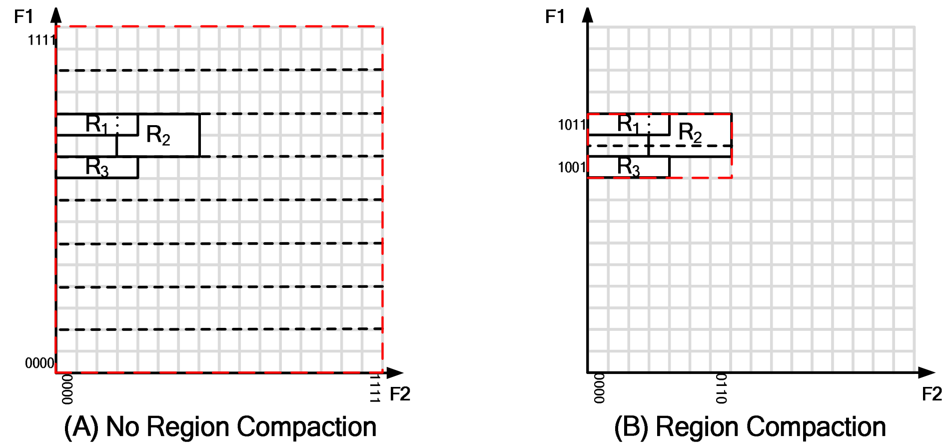


Fig. 3.5. Region division with and without region compaction.

understood. Fig. 3.5 illustrates two methods of dividing a region defined by two fields in a way that none of the resulting sub-regions contain more than two rules. The method shown in Fig. 3.5 (A) does this by performing eight cuts along the full length of field *F1*, with all resulting sub-regions containing two or less rules. This method of dividing the region allows for a simple scheme to be used when deciding which sub-region a packet should traverse to, with only two pieces of information required for each field. This information includes the number of cuts that need to be performed to each field of a packet header and the bits in these fields where the cuts need to be performed. A packet with a header value <u>101</u>1 for field *F1* will use its three MSBs to represent the index of the sub-region that must be selected as it is the first time that this region is cut. There are eight cuts to be performed, meaning that three bits are needed to represent the eight possible sub-regions that could be selected.

Performing eight cuts to the full length of field *F1* is wasteful in this example as the three rules that must be divided only span a small length of field *F1*. The region compaction heuristic used by HyperCuts overcomes this problem and is illustrated in Fig. 3.5 (B). As mentioned, region compaction only cuts the area covered by the rules and not the full region. Fewer cuts may therefore be needed to divide the region in a way that results in none of the sub-regions containing more than two rules. In this example region compaction reduces the number of cuts that are needed to divide the region from eight to two. The use of region compaction requires three pieces of information to be stored for each field in order to calculate the correct sub-region that must be traversed to. This

information includes the minimum and maximum limits of the compacted region for a given field (F_{min} and F_{max}) and the number of cuts (nc) that must be performed to this field in a packet's header (F_{header}). Equations 3.2 and 3.3 show how the index for each field is calculated.

$$((F_{\max} - F_{\min}) + 1) / nc = d \tag{3.2}$$

$$\lfloor (F_{header} - F_{\min}) / d \rfloor = index \tag{3.3}$$

A packet with the header value 1011 for field *F1* will have its index calculated as follows. $F1_{index} = \lfloor (11-9)/1.5 \rfloor = 1$ where the denominator $d = ((11-9)+1)/2 = 1.5$. This index is the sub-region that must be traversed to as only field *F1* is used for cutting. Use of the region compaction heuristic used in HyperCuts can lower memory consumption by reducing the number of sub-regions that need to be stored. It is not, however, suitable for hardware implementation as extra logic is needed to carry out the floating point division, which is required when calculating the sub-region that must be selected. The delay caused by the extra logic and additional clock cycles needed for floating point division will slow down the hardware accelerator, decreasing throughput and increasing power consumption.

Compacting of a Region through Pre-Cutting

A new method for compacting the region to be cut at each internal or root node in the decision tree through pre-cutting of the hyperspace is presented here. It uses the same methods employed by the scheme that uses no region compaction when calculating the sub-region a packet should traverse to. This scheme only requires an internal or root node to store the number of cuts that must be performed to each field of a packet header and the bits in these fields where the cuts need to be performed. The simplicity of this scheme helps to improve throughput and decrease power consumption. The region that needs to be divided is compacted by recursively cutting all fields in two. This cutting of a specific field in two stops and will not be carried out if it results in rules being contained in more than one sub-region. Each pre-cut to a field used to divide the region will halve the number of sub-regions that need to be stored and the number of cuts that need to be performed to a packet header when selecting the correct sub-region to traverse to. Each pre-cut to a field also means that the bits which need to be inspected in that field of a packet's header are shifted to the right one place.

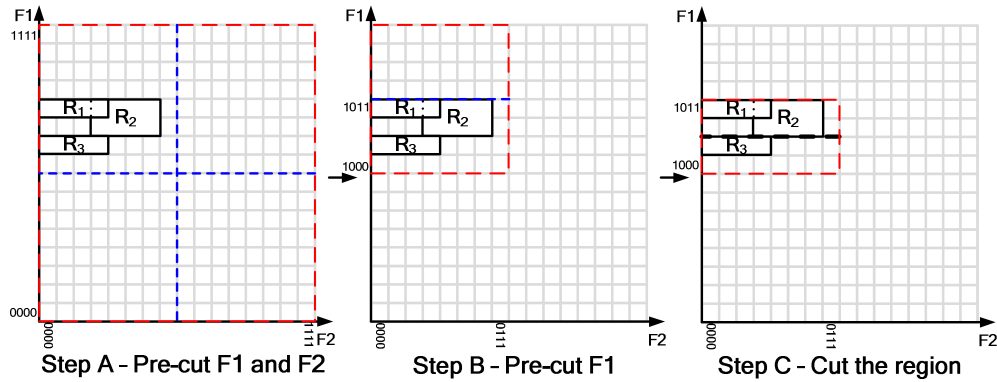


Fig. 3.6. Compacting of a region through pre-cutting.

Fig. 3.6 shows an example where pre-cutting is used to compact a region so that it can be cut more efficiently. The process begins by performing pre-cuts to field *F1* and *F2* as shown in step A, reducing the area that needs to be considered for cutting by 75%. Pre-cuts can be performed to both fields as it results in only one sub-region that contains rules. In step B only field *F1* is pre-cut as pre-cutting both fields *F1* and *F2* would result in more than one sub-region that contains rules. Pre-cutting field *F1* in step B reduces the area that needs to be considered for cutting by another 50%. Finally, in step C no more pre-cuts can be performed so the compacted region is split in two, with none of the resulting sub-regions containing more than two rules. Pre-cutting gives the same effect as the region compaction heuristic used by HyperCuts in this example, with the number of sub-regions that need to be stored reduced from eight to two when compared to the method where no form of region compaction is used.

A packet with a header value of 10<u>1</u>1 for field *F1* can calculate the sub-region that it must traverse to by simply using its third MSB as an index. The two MSBs are ignored because field *F1* has been pre-cut twice. Only the third MSB is required as an index as only two cuts are performed to this field, meaning that one bit can represent both possible sub-regions that could be selected.

3.3.3 Rule Storage

Some slight modifications have also been made to the way that a rule is stored in a leaf node to reduce both memory consumption and the number of memory accesses needed to retrieve the information required to match a packet header to a rule. The first modification is to store the actual rule in the leaf node rather than a

Prefix Length	Bit[34:7]	Bit[6:3]	Bit[2:1]	Bit[0]
32	32-Bit IP		00	0
31	32-Bit IP		01	0
30	32-Bit IP		10	0
29	32-Bit IP		11	0
28	28-Bit IP	011100		1
27	28-Bit IP	011011		1
:				
1	28-Bit IP	000001		1
0	28-Bit IP	000000		1

Fig. 3.7. Encoding scheme used for source and destination IP address.

pointer to the rule. This was found during testing of the ACL, FW and IPC rulesets created using ClassBench to have only a small increase in memory consumption for some rulesets and a reduction in memory consumption for others as pointers to rules do not need to be stored. Storing the actual rule rather than a pointer to it allows for a large increase in throughput as data is presented to the hardware accelerator one clock cycle earlier.

A second modification is to reduce the amount of bits required to store the source and destination IP address from 76 bits down to 70 by using an encoding scheme. An IP address usually requires 32 bits to store its address and 6 bits to store its mask. The mask number is used to specify the number of MSBs of the address that must be an exact match to the corresponding bits in a packet header to record a match. The remaining LSBs are wildcard bits, meaning that the value of the corresponding bits in a packet header can have any value and still record a match. The encoding scheme stores the 32-bit IP address and 6-bit mask as a 35-bit number. The lowest bit is used to indicate if more than 28 bits of the IP address need to be matched exactly. If not set, 32 bits are used to store the IP address, with the remaining two bits indicating the actual number of bits that need to be matched. If set, 28 bits are used to store the IP address, with the remaining 6 bits indicating the actual number of bits that need to be matched. The encoding scheme used by the hardware accelerators is shown in Fig. 3.7. This method of encoding the IP address and mask can easily be modified so that only 33 bits are needed, with only a slight increase in the logic needed to decode the information.

Each rule will require 143 bits to record the information needed to match it to a packet header, with the source and destination IP address each requiring 35 bits. The source and destination port numbers both require 32 bits, with each port number's minimum and maximum range values needing 16 bits. A total of 9 bits

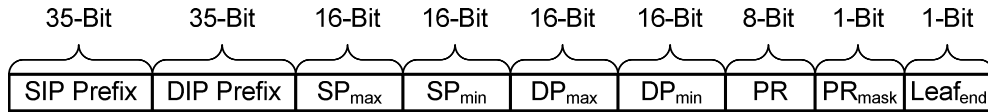


Fig. 3.8. Layout of information needed to match a packet header to a rule.

are required to store the information needed to match the protocol number, with 8 bits used to store the protocol number and 1 bit to store its mask. The mask only requires 1 bit as the protocol number can only be an exact match or wildcard. Each rule also has a flag bit that is set if it is the last rule in a leaf node. The packet classifier will know that it has finished searching a leaf node if it comes across a rule with this bit set. Fig. 3.8 shows the memory layout of the information needed to match a packet header to a rule.

3.4 Cut Selection

The cutting information for each field consists of two pre-computed values. The first pre-computed value is called *Cuts* and it is used to indicate how many cuts can be performed on a given field. The number of cuts that can be performed on a given field is limited to be a power of two for ease of implementation. An 8-bit protocol number limited to 256 cuts, for example, can only have 0, 2, 4, 8, 16, 32, 64, 128 or 256 cuts performed to it. To save space a 4-bit number for *Cuts* can be used to represent the nine different cut values. The number of bits needed to store the *Cuts* value for a field can be calculated using Equation 3.4 where *MaxCuts* is the maximum number of cuts allowed to the field.

$$NumberOfBits = \lfloor \log_2(\log_2(MaxCuts)) \rfloor + 1 \tag{3.4}$$

The value of *Cuts* is also the length of the bit-mask for a given field. This bit-mask will be ANDed with the appropriate bits of a packet field to extract the index for this field. Before the index of the child node is calculated, the *Cuts* information is extended to form the bit-mask for each field. The second pre-computed value for each field is called *BPos*, and it is used to indicate the bits that the bit-mask should be ANDed with. In the calculation of a child node index, *BPos* is the number of lower bits in a packet field that need to be removed by shifting the field right, before the operation of ANDing with the bit-mask can be performed. The protocol number, for example, will require three bits to store its *BPos* value as it will need to be shifted right 0-7 places. The number of bits needed

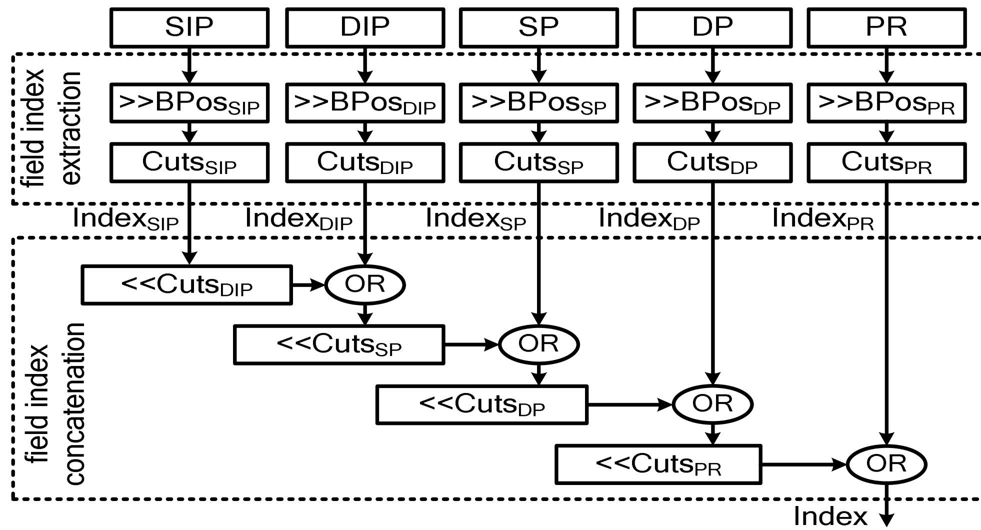


Fig. 3.9. Architecture of cut selection logic.

to store the *BPos* value for a given field is calculated using Equation 3.5 where *LengthOfField* is the number of bits used to store the field.

$$NumberOfBits = \log_2(LengthOfField) \quad (3.5)$$

The architecture of the cut selection logic is shown in Fig. 3.9. It can calculate the appropriate child node that a packet must traverse to in a single clock cycle as a result of the simplicity of the new region compaction and cutting schemes that have been presented here. These schemes can generate the appropriate child node index by performing simple shift and AND operations. The shifting of bits is carried out using multiplexers so that all shift operations can be performed in a single clock cycle. The child node index is generated in two stages. The first stage generates the sub-index for each field, while the second stage concatenates these sub-indexes together to form the final index of the child node to be selected. The sub-index for a field is generated by first shifting it to the right. The *BPos* value for the field specifies how many bits it should be shifted. This shifted value is ANDed with the bit-mask for the field to create its sub-index. As previously mentioned, the bit-mask for a field is generated by extending its *Cuts* value. The sub-indexes are concatenated in the final stage to form the final index of the child node by left shifting the sub-index of each field by the length of the sub-index of the next field and then ORing them together. This is done until the $index_{PR}$ is combined with the others as illustrated in Fig. 3.9. The length of the sub-index of each field is specified by its *Cuts* value.

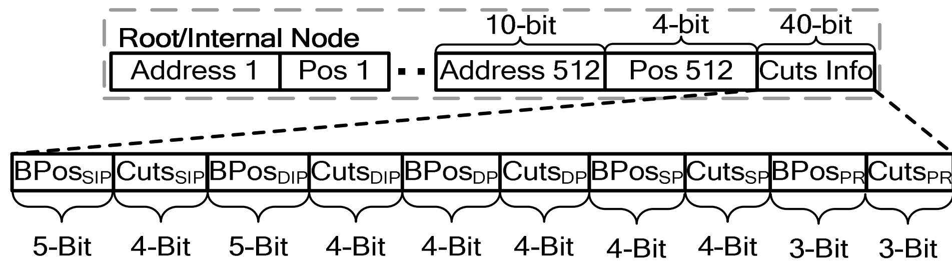


Fig. 3.10. Layout of root/internal node when using ultra-wide memory.

3.5 Memory Organisation

This section explains how to store the root, internal and leaf nodes of the decision tree search structures built for the architectures designed to use ultra-wide and reduced width memory words. It also explains how the nodes are carefully arranged in memory after the decision tree has been built to ensure that there are as few gaps of unused memory as possible.

3.5.1 Ultra-Wide Memory Words

The architecture that uses ultra-wide memory employs 7,704-bit wide memory words that can hold all the information needed to store a root or internal node, or up to 48 rules of a leaf node. This helps to reduce the number of memory accesses required to classify a packet. It was found through testing that 7,704-bit wide memory words offered the best trade-off in terms of number of memory accesses needed to classify a packet and maximum obtainable clock speed when using ClassBench generated rulesets that contained a large number of wildcard rules. A root or internal node can perform a maximum of 512 cuts when dividing the hyperspace, limiting the number of bits required to store them to 7,208. Fig. 3.10 shows the layout of a root/internal node that contains the maximum allowed number of 512 child nodes. Each cut uses 14 bits to store the pointer to its child node. These pointers use 10 bits to store the memory address of their child node, with an address of zero indicating that the child node is empty and no matching rule has been found. The remaining 4 bits are used to indicate the starting position of the child node in a memory word and whether it is an internal or leaf node. All internal nodes are stored at the start of a memory word, while a leaf node can have one of 15 possible starting positions. This gives one memory word the ability to store up to 15 different leaf nodes.

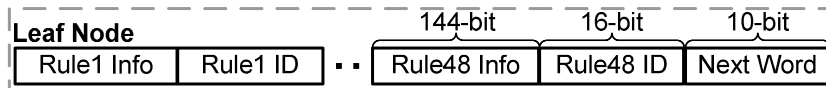


Fig. 3.11. Layout of leaf node when using ultra-wide memory.

An internal or root node requires 40 bits to store its cutting information. This information is made up of the *Cuts* and *BPos* values for each field. The source and destination IP addresses and port numbers are each limited to 512 cuts, meaning that 4 bits are adequate to store the *Cuts* value for each of these fields. This is because a limit of 512 cuts allows only ten possible cut values. A total of 3 bits are required to store the *Cuts* value for the protocol number as it has nine possible cut values. Each 32-bit source and destination IP address will require 5 bits to store its *BPos* value as they can be shifted 0-31 places. The 16-bit source and destination port numbers require 4 bits to store this value as they can be shifted 0-15 places, while the 8-bit protocol number requires 3 bits as it can only be shifted 0-7 places. Each packet being classified needs to pass through the root node. The root node is therefore stored in a register separate from main memory to reduce the number of memory accesses needed to classify a packet.

A total of 160 bits are required to store each rule contained within a leaf node, of which 16 bits are used to store a rule's ID, 143 bits to store the information needed to match a rule to a packet header and 1 bit to indicate if it is the last rule stored in a leaf node. Each memory word storing a leaf node will also store the address of the next memory word. This address is used when retrieving the rules of a leaf node that cannot fit in a single memory word. Storing this address reduces logic as the starting address of a leaf node spanning multiple memory words does not need to be stored and incremented when retrieving its rules. Fig. 3.11 shows the layout a leaf node with 48 rules stored in a single memory word.

In order to reduce memory consumption the nodes are rearranged after the search structure has been built. All the internal nodes are stored first, followed by the leaf nodes. This means that the leaf nodes can be saved contiguously in the search structure, thus improving the storage efficiency of rules. The HyperCuts algorithm uses parameters known as *spfac* and *binth* to trade off throughput against memory consumption. A parameter introduced here to trade throughput against memory consumption for the architecture that uses ultra-wide memory words is called

speed. The leaf nodes are packed one after another as tightly as possible when the speed parameter is not set. This means that the search structure is saved in the most memory efficient way possible but will not result in the highest possible throughput, with the number of memory accesses needed to classify a packet calculated using Equation 3.6.

$$MemoryAccesses = \lfloor ((z + pos)/48) + 1 + x \rfloor \tag{3.6}$$

Where the number of internal nodes traversed to reach the leaf node is represented by x. The starting position of the leaf node in a memory word is represented by *pos*, and z is the position of the matching rule in the leaf node. The *speed* parameter being set will mean that a leaf node is only stored in a memory word with a staring position greater than zero if Equation 3.7 is satisfied:

$$RulesStoredInLeaf + pos \leq 48 \tag{3.7}$$

This means that there may be reduced storage efficiency as the leaf nodes might no longer be stored as tightly as possible. Reduced storage efficiency will, however, lead to an increase in throughput as the number of memory accesses needed to classify a packet will be calculated using Equation 3.8. The starting position of the leaf node *pos* will have no effect on the number of memory accesses needed to classify a packet, meaning that it can be removed.

$$MemoryAccesses = \lfloor (z/48) + 1 + x \rfloor \tag{3.8}$$

3.5.2 Reduced Width Memory Words

Two similar architectures that use reduced width memory words are presented here. The first architecture uses the internal memory of an FPGA, exploiting the flexibility of this internal memory by using 324-bit wide memory words. The second architecture is designed to use external memory and is limited to using 288-bit wide memory words due to the rigidity of external data bus widths.

Internal Memory

The architecture that uses internal memory requires 45 bits to store the *Cuts* and *BPos* values used to cut the root node. This cutting information is again placed in a register separate from main memory as it must be accessed by every packet being classified. As with the previous architecture this reduces the number of memory accesses needed to classify a packet by one. The source and destination IP

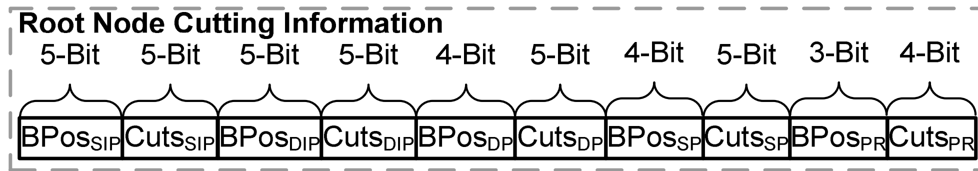


Fig. 3.12. Layout of root node cut information when using reduced width memory.

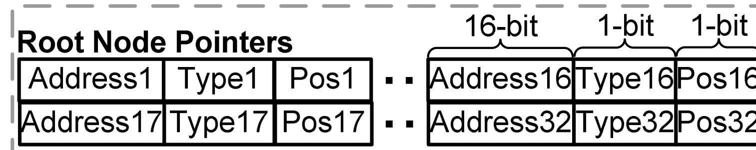


Fig. 3.13. Layout of root node pointers when using reduced width internal memory.

addresses have been limited to performing a maximum of 262,144 cuts to the hyperspace as explained in Section 3.3.1. This means that 5 bits are required to store their *Cuts* value as there are 19 possible cut values. The source and destination port numbers can perform up to 65,536 cuts when dividing the hyperspace, which means that they also require 5 bits to store their *Cuts* value as they have 17 possible cut values. The protocol number requires 4 bits to store its *Cuts* value as it has nine possible cut values. The remaining 21 bits are used to store the *BPos* values used for indicating which bits of the five fields should be used to form the child node index. Fig. 3.12 shows the layout of the root node cutting information.

The root node requires 18 bits to store each of its child node pointers. This means that each memory word can hold 16 pointers, with the MSBs of the child node index used to retrieve the memory word where its pointer is stored and the LSBs used to indicate its position in that memory word. The pointer uses 16 bits to store the child node's address in memory, with a value of zero again meaning that the child node is empty and no matching rule has been found. Another bit is used to indicate if the child is an internal or leaf node, while the final bit indicates the starting position of the node at its memory location if it is a leaf node. This bit is required because each memory word can hold two rules. Fig. 3.13 shows the layout of a root node's pointers. In this example the root node has 32 child nodes, with the pointers to these nodes occupying two memory words.

Each internal node requires 36 bits to store its cutting information. The only difference in the cutting information for an internal node and the root node is that the *Cuts* value for all five dimensions only requires 3 bits as the number of cuts to

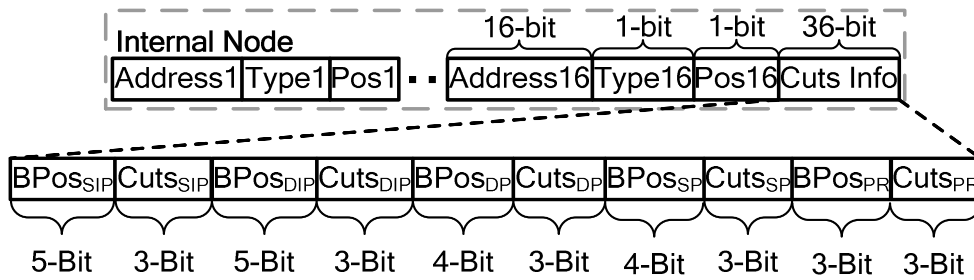


Fig. 3.14. Layout of internal node when using reduced width internal memory.

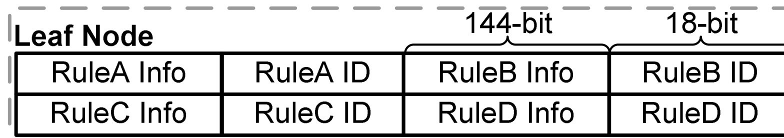


Fig. 3.15. Layout of leaf node when using reduced width internal memory.

each dimension has been limited to 16, meaning that there are only five possible cut values for each field. An internal node can therefore fit fully in one memory word as the cutting information and maximum of 16 pointers will require 324 bits to store. Fig. 3.14 shows the layout of an internal node with the maximum allowed number of 16 child nodes, while Fig. 3.15 shows the layout of a leaf node containing 4 rules, with these rules stored across two memory words. Each rule in a leaf node for this architecture requires 162 bits, of which 18 bits are used to store the rule ID, 143 bits to store the information needed to match a rule to a packet header and 1 bit to indicate if the rule is the last rule stored in a leaf node.

The nodes that form the decision tree are again rearranged after it has been built in order to obtain maximum storage efficiency, with the rearranging carried out carefully so that no extra memory accesses are added to the worst case required to classify a packet. The pointers of the root node’s child nodes are stored first, followed by the internal nodes. Leaf nodes that contain an even number of rules are stored next and then the leaf nodes that contain an odd number of rules. This will ensure that there are no gaps of unused memory, with no extra memory accesses added to the worst case when searching a leaf node. This is because each memory word stores two rules.

External Memory

The architecture that uses external memory employs 288-bit wide memory words. This allows the use of cheap Double Data Rate 2 Synchronous Dynamic Random

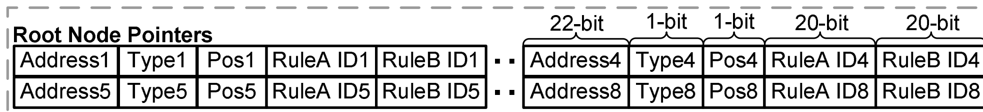


Fig. 3.16. Layout of root node pointers when using reduced width external memory.

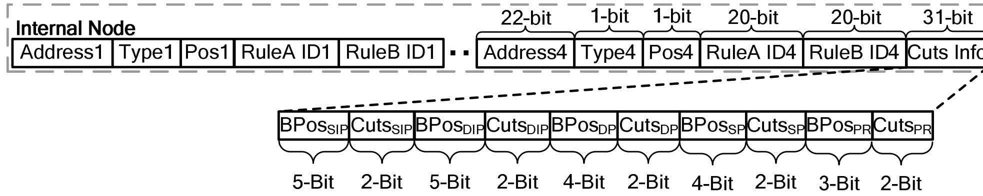


Fig. 3.17. Layout of internal node when using reduced width external memory.

Access Memory (DDR2 SDRAM) reading in 72-bit wide memory words in bursts of four. As with the previous architecture, the information required to perform cuts to the root node requires 45 bits, with this information stored in a separate register to reduce memory accesses. This time the pointers for the root node require 64 bits, with 22 of these bits used to give the memory address where the child node is stored. A value of zero again indicates an empty child node and no match. The pointer also includes a bit that indicates the node type and another bit that gives the node's starting position in a memory word if it is a leaf node. Each pointer also stores 20-bit rule IDs for the first two rules that could be stored if the child node is a leaf. This is done because the 288-bit wide memory words are only wide enough to store the rule information used for comparison and not the rule ID. A packet matching one of the first two rules in a leaf node will not require an extra memory access to retrieve its rule ID. A memory word can hold a maximum of four pointers due to the large amount of information that a pointer stores. Fig. 3.16 shows the layout of a root node's pointers. The root node in this example has eight child nodes, with the pointers to these nodes occupying two memory words.

The cutting scheme used for the internal nodes has again been designed so that all information needed to traverse them can fit fully in a single 288-bit wide memory word. Each internal node is allowed a maximum of four child nodes, with the pointer for each using 64 bits. The cutting information has been reduced to 32 bits as each field can perform a maximum of four cuts, which means that 2 bits are required to store the *Cuts* value for each field as each field has only three possible cut values. The layout of an internal node is shown in Fig. 3.17. This example shows an internal node with the maximum allowed number of 4 child nodes.

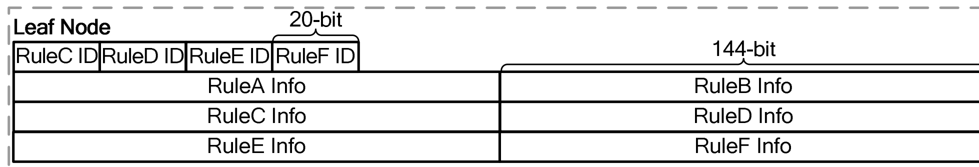


Fig. 3.18. Layout of leaf node when using reduced width external memory.

Each rule in a leaf node requires 143 bits for its comparison information as explained previously, and an additional bit to indicate if it is the last rule stored in a leaf node. The leaf node will come in two parts if the number of rules stored exceeds two. The first part will store the rule ID for the third and subsequent rules in the leaf node, with each memory word storing up to eight 20-bit rule IDs. The second part stores the rule comparison information. Fig. 3.18 shows an example of how a leaf node is laid out. This example shows a leaf node containing six rules with the information needed to match a rule to a packet header and the rule IDs stored across four memory words. It can be seen that the leaf node only needs to store the rule IDs for the third and subsequent rules. This is because the rule IDs for the first two rules are stored with the pointer to the leaf node.

The starting position of the rule comparison information is given as the address of the leaf node. One of the first two rules matching will mean that a memory access for the rule ID will not be needed as this information was already given in the leaf node pointer. A counter is used to count how many memory accesses were required for rule comparison information before a match takes place. The MSBs of this counter are subtracted from the leaf node's starting address when a match takes place. This gives the memory address of the matching rule ID. The location of the matched rule and LSBs of the counter are used to locate the position of the matching rule ID on this memory word.

This search structure is compacted to ensure that there are as few gaps of unused memory as possible by first storing the pointers of the root node, followed by the internal nodes. All leaf nodes that contain two or more rules are then stored. The final step involves storing the leaf nodes that contain a single rule in places where the memory would otherwise not have been used. This is done to plug as many gaps as possible. These gaps are located in memory words used to store less than eight rule IDs and at the end of leaf nodes that contain an odd number of rules.

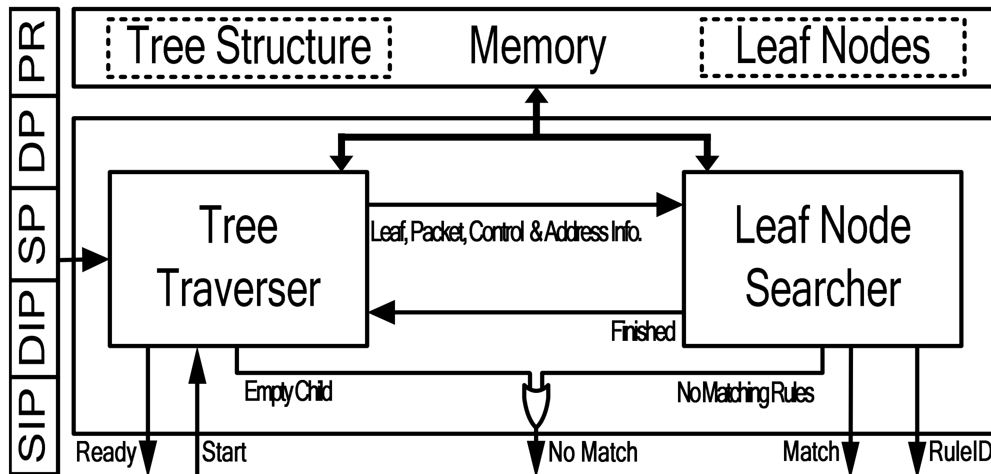


Fig. 3.19. Block diagram of the architecture used by the packet classification engines.

3.6 Packet Classification Engine

Fig. 3.19 shows a block diagram of the architecture used by the packet classification engines designed to use on-chip ultra-wide memory, on-chip reduced width memory and external reduced width memory. All three engines presented are built using two main modules. The first module is a tree traverser that is used to traverse a decision tree using header information from the packet being classified. The decision tree is traversed until an empty child node is reached, meaning that there is no matching rule, or until a leaf node is reached. A leaf node being reached will result in the tree traverser passing the packet header and information on the leaf node reached to the second module known as the leaf node searcher. The leaf node searcher compares the packet header to the rules contained within the leaf node traversed to until either a matching rule is found or the end of the leaf node is reached with no rule matched. The leaf node searcher in all three engines presented employs multiple comparator blocks that work in parallel. This allows the searching of more than one rule on each memory access, reducing lookup times.

Information on the decision tree's root node is stored in registers in the tree traverser for all three engines. This makes it possible for the tree traverser to begin classifying a new packet, while the previous packet is being compared to rules in the leaf node it traversed to for a matching rule using the leaf node searcher. This use of pipelining allows for a maximum throughput of one packet every clock

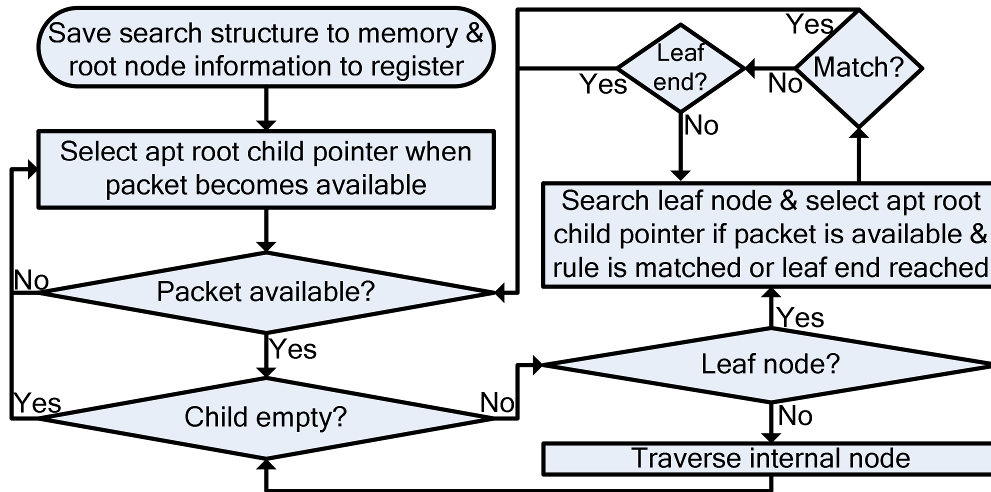


Fig. 3.20. Operation of engine using ultra-wide memory words.

cycle when the packet classification engine designed to utilise ultra-wide memory words is used. A maximum throughput of one packet every two clock cycles can be achieved when using the packet classification engine designed to use on-chip reduced width memory.

3.6.1 Architecture of Engine Using Ultra-Wide Memory Words

The flow chart shown in Fig. 3.20 explains the operation of the packet classification engine designed to use on-chip ultra-wide memory. The engine has been designed to traverse the root node of a decision tree without requiring any memory accesses. This is possible because its tree traverser can hold all the information needed to traverse the root node in registers. This can be done because of the limited number of cuts allowed to the root node when this engine is used, reducing the amount of information that needs to be stored. The information stored is the root node's cutting information and its child pointers, of which there can be a maximum of 512. All internal nodes require one memory access to be traversed, while a leaf node can be searched at a rate of 48 rules per memory access. This makes it possible for the packet classification engine to classify a packet in one memory access at worst when the decision tree is made up of only root and leaf nodes, with each leaf node storing no more than 48 rules.

To classify a packet, the search structure is first saved to memory. The root node cutting information is also registered to the *R Node Cut Data* register and the pointers to the root's child nodes are stored in the *R Node Pointers* register. These

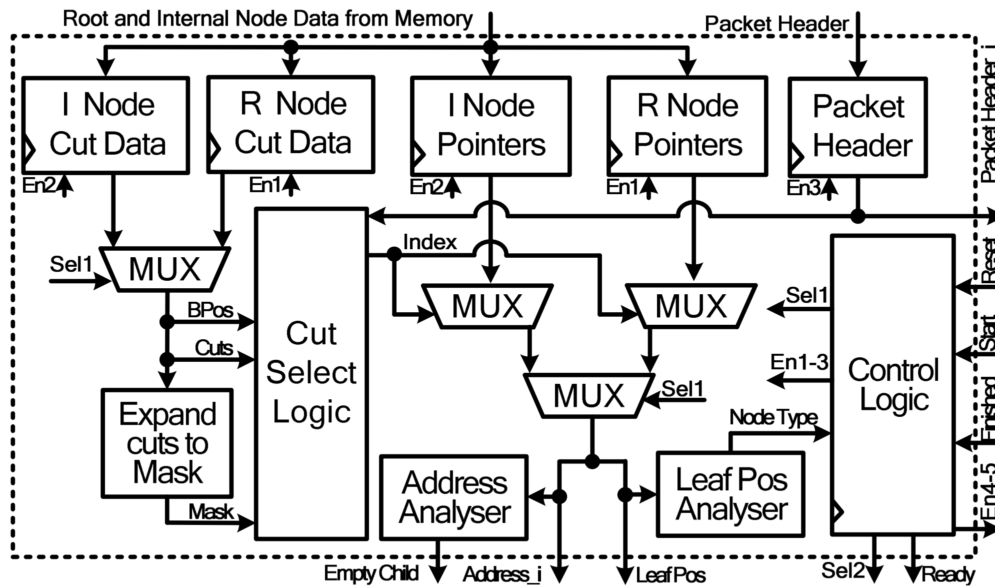


Fig. 3.21. Architecture of tree traverser using ultra-wide memory words.

registers can be seen in Fig. 3.21, which shows the architecture of the tree traverser module. The tree traverser begins by monitoring a *Start* signal from a small packet buffer used to store the fields of a packet's header that are needed for classification. This *Start* signal notifies the classification engine when there are packets available to be classified. The tree traverser will load a packet header from the buffer to its *Packet Header* register when it becomes available for classification and will assert a *Ready* signal to notify the buffer that it has loaded the header. The header and root node cutting information are used to calculate the index of the child node that should be traversed to. This index picks which of the root's child pointers should be selected from the *R Node Pointers* register. The pointer selected stores the node's type, address in memory and starting position in the memory word located at this address if a leaf node is traversed to.

The node's address is analysed to check if it is an empty node. The address will be zero if an empty child node has been reached and no matching rule has been found. In this case the classification engine will assert an *Empty Child* signal to indicate that no matching rule has been found and go back to scanning the buffer for more packets to be classified. A value greater than zero means that the node is not empty. In this case the value is analysed that indicates the node's type and starting position in the memory word where it is located to see if the node to be traversed to is a leaf or internal node. All the steps required to traverse the root

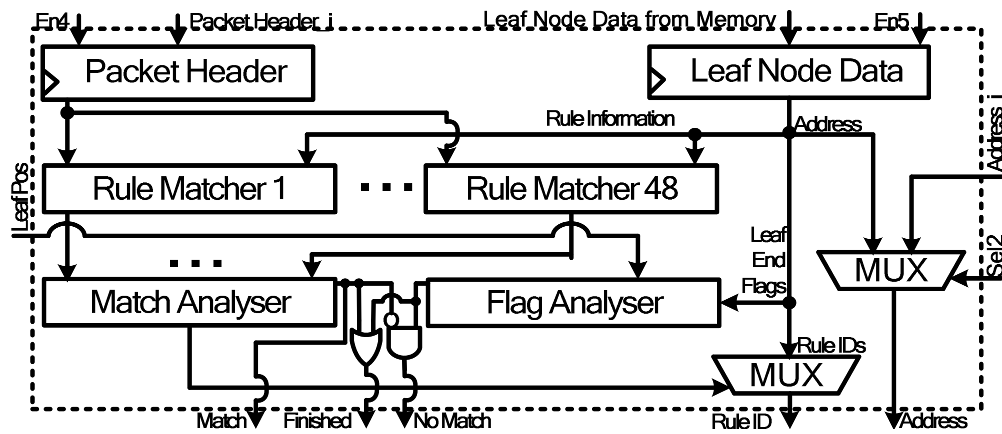


Fig. 3.22. Architecture of leaf node searcher using ultra-wide memory words.

node from registering the packet header to finding the address, type and starting position in a memory word of the child node to be traversed to is carried out in one clock cycle.

An internal node being traversed to will require the loading of the internal node's cutting information to the *I Node Cut Data* register and its pointers to the *I Node Pointers* register. The same tasks used to traverse the root node are performed to the packet header stored in the *Packet Header* register using the registered internal node information. These tasks involved finding out if the child node to be traversed to is an empty node and if not its address, its type and starting position in the memory word where it is located. The traversal of an internal node also requires one clock cycle.

A leaf node being reached will mean passing the header belonging to the packet being classified and the leaf node's starting position in the memory word where it is located to the leaf node searcher, whose architecture is shown in Fig. 3.22. The leaf node searcher registers this packet header to the *Packet Header* register. The leaf node data returned from memory is stored using the *Leaf Node Data* register. This data is made up of the information required to compare up to 48 rules and their rule IDs. The leaf node searcher uses 48 separate comparator blocks that work in parallel to compare the rule information loaded with the packet header to be classified. The output of each comparator is checked to see if a match has been found. No rule being matched will mean checking the flag bit of the rules loaded to determine if the last rule in the leaf node has been reached. No match and the last rule in the leaf not being reached will mean using the address stored in the

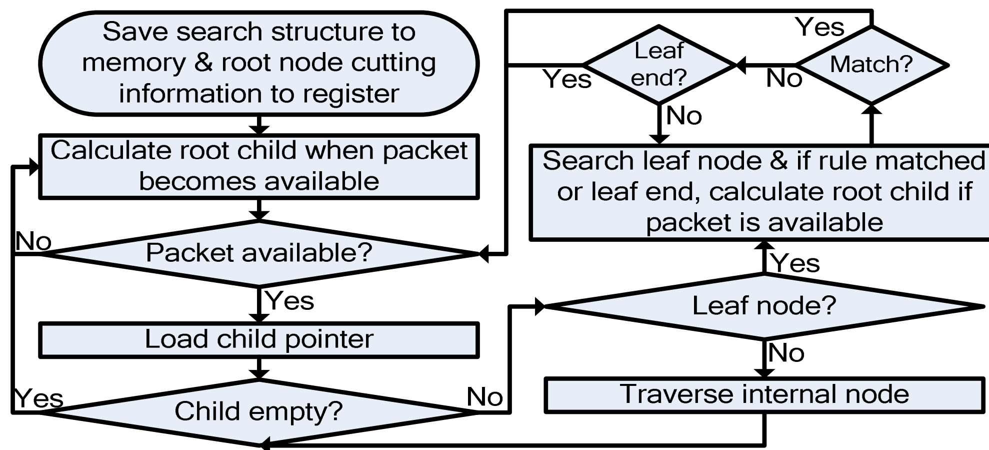


Fig. 3.23. Operation of engine using reduced width internal memory.

current memory word being analysed to load the memory word containing the next 48 rules. This process is continued until a matching rule is found or the last rule in the leaf has been reached and no match found. A matching rule being found will mean asserting a *Match* signal and outputting the appropriate rule ID. No matching rule being found will mean asserting a *NoMatch* signal.

The tree traverser module is able to traverse the root node for a new packet if there are packets available to be classified while the leaf node searcher is searching the leaf node of the previous packet. The tree traverser asserts its *Ready* signal and loads a new packet to its *Packet Header* register and repeats the steps required to traverse the root node. The leaf node searcher can then begin searching the leaf node for this packet as soon as it has finished searching the leaf node of the previous packet. This means that it is possible to classify a packet on every clock cycle when the decision tree only contains a root and leaf nodes, with a leaf node containing no more than 48 rules. The packet classification engine will remain in an idle state if there are no packets to be classified, where it continuously monitors the buffer's *Start* signal. It does this until a packet becomes available for classification, in which case it will assert its *Ready* signal and repeat the process described all over again to classify the new packet.

3.6.2 Architecture of Engines Using Reduced Width Memory Words

The flow chart shown in Fig. 3.23 explains the operation of the packet classification engine designed to reduced width internal memory. The engine has been designed to traverse a root or internal node in one memory access. It can also

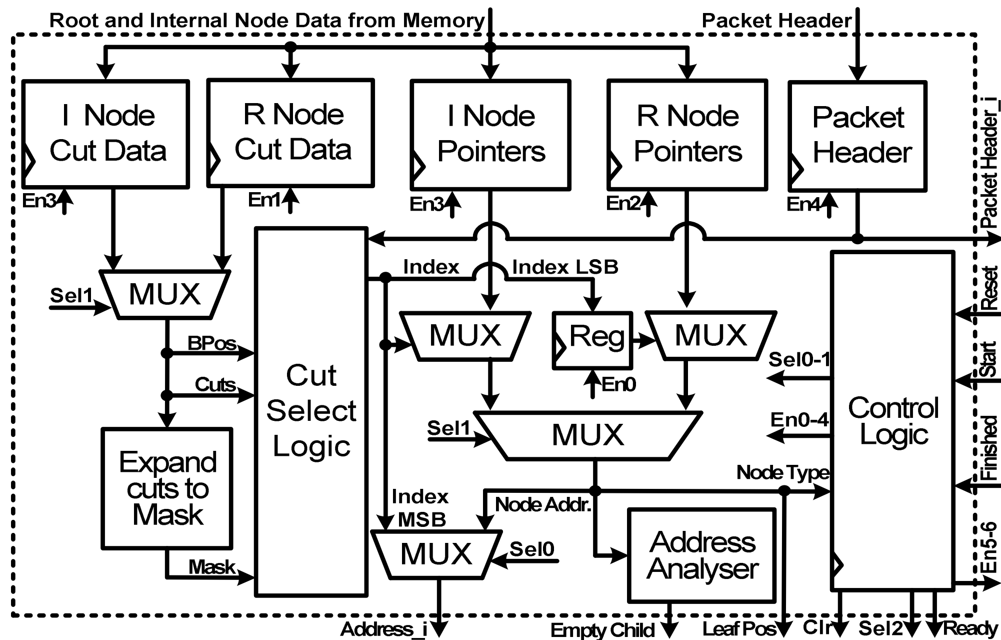


Fig. 3.24. Architecture of tree traverser using reduced width internal memory.

search leaf nodes at a rate of two rules per memory access. This makes it possible for the packet classification engine to classify a packet in two memory accesses at worst if the decision tree is made up of only a root node and leaf nodes storing no more than two rules.

A packet is classified by first saving the search structure to memory and registering the root node cutting information to the *R Node Cut Data* register located in the tree traverser, whose architecture is shown in Fig. 3.24. The tree traverser then communicates with a packet buffer used to store the required fields of a packet's header needed for classification in the same manner described in the explanation of the previous engine through the use of *Ready* and *Start* signals. The tree traverser loads a packet header to the *Packet Header* register when it becomes available for classification. The header loaded and the stored root node cutting information are used to calculate the child node that should be traversed to. Loading the packet header and calculating which of its child nodes should be traversed to takes one clock cycle. The MSBs of the child node index calculated are used to load the memory word containing the child node's pointer on the next clock cycle.

On this clock cycle the memory word containing the correct child node pointer is registered to the register labelled *R Node Pointers*. The LSBs of the child node

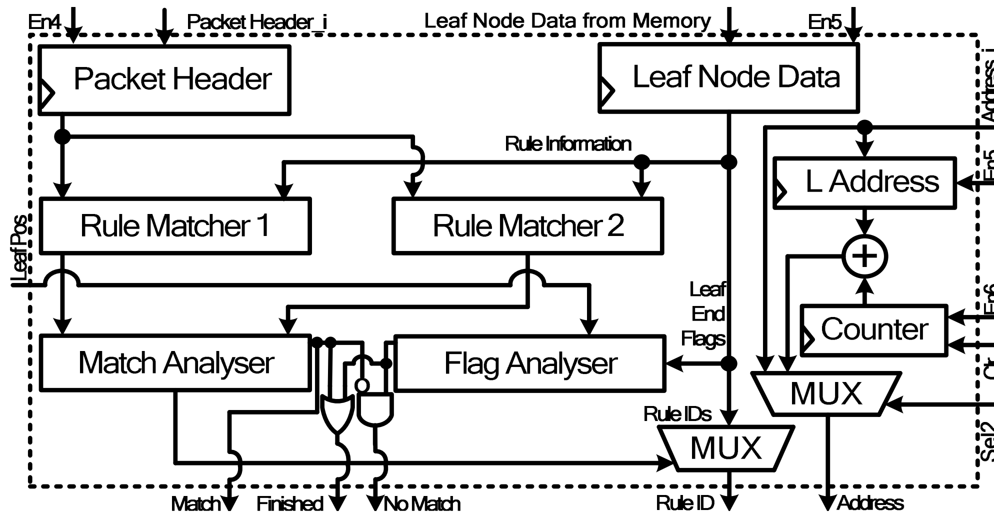


Fig. 3.25. Architecture of leaf node searcher using reduced width internal memory.

index are used to select the pointer in this memory word that should be selected. The address of the child node being traversed to is analysed to check if it is zero. An *Empty Child* signal is asserted if it is zero to indicate that no matching rule has been found and that the classification engine should begin the process of classifying a new packet. An address greater than zero will mean examining the bit indicating the node type to see if the node to be traversed to is a leaf or internal node. Loading and analysing the root node pointer takes one clock cycle.

In the case where an internal node is to be traversed to, the memory word loaded on the next clock cycle will contain the internal node's child pointers and cutting information. These child pointers are stored using the *I Node Pointers* register, while the cutting information is stored using the *I Node Cut Data* register. The cutting information is again used to calculate which of the internal node's child nodes is to be traversed to, with this index used to select which of its pointers loaded is to be analysed. The internal node pointer information is analysed in the exact same way as the root node pointer information. Traversing an internal node takes one clock cycle.

A leaf node being traversed to will mean using the leaf node searcher whose architecture is shown in Fig. 3.25 to search the leaf for a matching rule using the steps described by the packet classification engine that uses the ultra-wide memory words. One difference is that this leaf node searcher can only compare two rules per memory access due to the reduced width memory words. Another

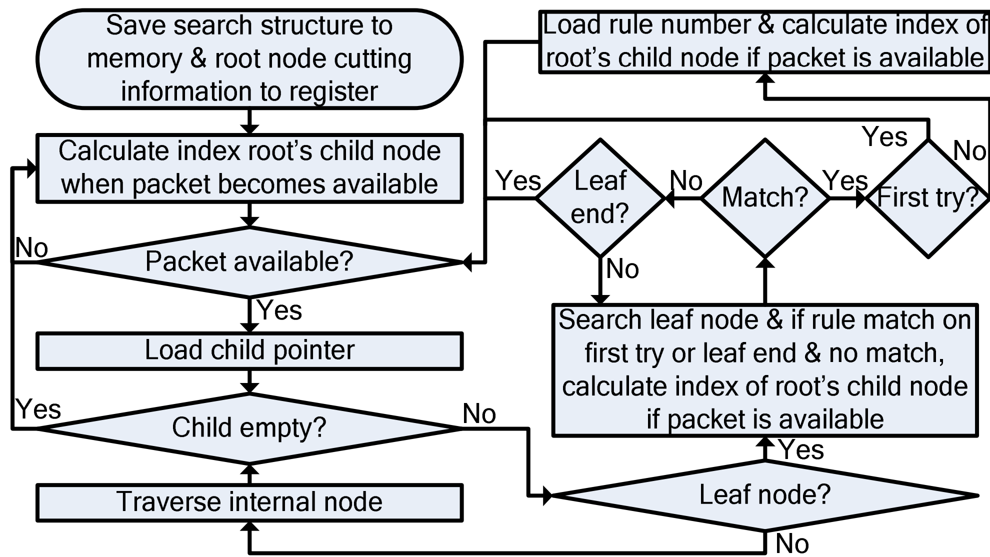


Fig. 3.26. Operation of engine using reduced width external memory.

difference is that the reduced width memory words do not have enough space to store the address of subsequent memory words, which may need to be fetched when retrieving the rules belonging to a leaf node. These addresses must be generated by the leaf node searcher using a counter that increments the leaf node's starting address.

The tree traverser module is again able to begin the process of classifying a new packet if there are packets available to be classified while the leaf node searcher is searching the leaf node of the previous packet. The tree traverser loads a new packet to its *Packet Header* register and uses the root node's cutting information stored in the *R Node Cut Data* register to calculate the index of the child node that must be traversed to. The pointer belonging to this child can be returned from memory as soon as the leaf node searcher is finished with the previous packet.

External Memory

Finally, the last packet classification engine presented is the engine that uses reduced width external memory. Fig. 3.26 shows a flow chart explaining its operation. The architecture of the tree traverser used by this engine is almost identical to the one shown in Fig. 3.24. It traverses the root and internal nodes in the same way, with the only difference being that a child pointer will contain the rule IDs of the first two rules stored in the node it points to, if the node pointed to is a leaf node. These rule IDs are passed to the leaf node searcher, along with the

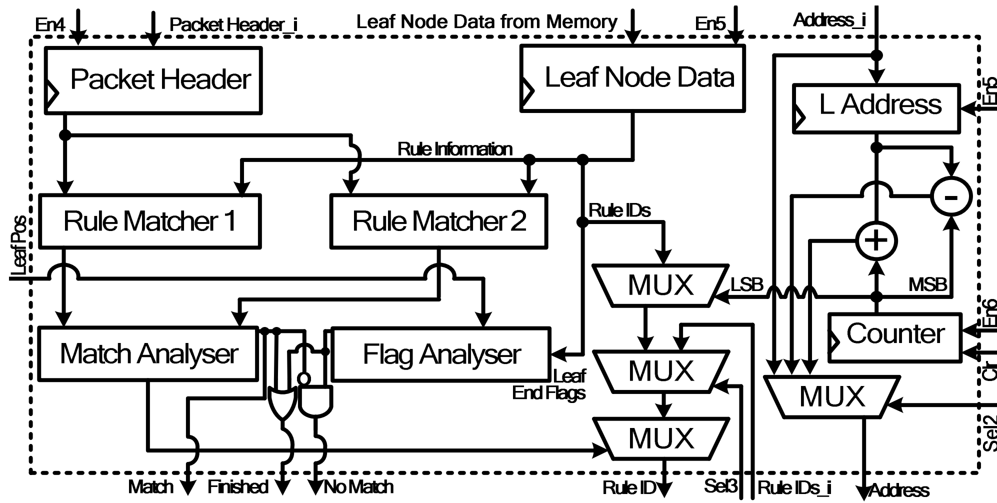


Fig. 3.27. Architecture of leaf node searcher using reduced width external memory.

packet header being classified, address of the leaf node to be searched and its starting position in the memory word located at this address. As explained, a pointer stores these rule IDs because the 288-bit wide memory words used by this architecture are only wide enough to store the comparison information needed for two rules and not their rule IDs. A match in one of the leaf node's first two rules will mean that a memory access is not required to retrieve the ID of the matching rule, while a matching rule not located in the first two rules will require an additional memory access to retrieve the appropriate rule ID.

The leaf node searcher used by the engine is shown in Fig. 3.27. It works by first registering the packet passed to it by the tree traverser in the *Packet Header* register. It also stores the leaf node information returned from memory in the *Leaf Node Data* register and the starting address of this leaf node in the register labelled *L Address*. Again two comparators are used in parallel to compare the rule information to the packet header being classified. A match in the first attempt will mean asserting the *Match* signal and using the appropriate rule ID from the leaf node's pointer, loaded previously. The *No Match* signal will be asserted if the end of the leaf node is reached and there is no matching rule. In either case a new packet can be loaded from the buffer if available on this clock cycle and its child index calculated using the root node's cutting information stored in the tree traverser. The root node's pointer will be loaded from memory on the next clock cycle if a new packet is available, otherwise the state where the classifier waits for a packet will be entered.

The leaf node's address will be incremented in the case where the leaf end is not reached and there is no matching rule, with the retrieved rules stored in the *Leaf Node Data* register. This address is generated by adding the leaf node's starting address to the value of a counter that increments each time another two rules need to be loaded. The next two rules in the leaf node are again compared to the packet header stored. This process continues until a match takes place or a leaf end is reached with no match. A leaf end reached with no match will mean loading a new packet if available, while a match will mean retrieving the matching rule ID. The address of the matching rule ID is generated by subtracting the MSBs of the counter from the starting address of the leaf node.

The memory word with the matching rule ID is loaded on the next clock cycle with the LSBs of the counter and matching rule position used to pick the correct rule ID from the memory word retrieved. During this cycle the *Match* signal will be asserted and the packet buffer checked for new packets that could be classified. Again, a packet being available will mean calculating the index of the child node the new packet must traverse to and loading its pointer on the next clock cycle, while no available packet will mean going to the state where the engine waits for a new packet to be classified. The engine's architecture makes it possible to classify a packet in two memory accesses at worst if the decision tree is made up of only a root and leaf nodes, where the leaf nodes store no more than two rules.

3.7 Configuration of Multiple Engines Operating in Parallel

The packet classification engines that use on-chip ultra-wide memory and reduced width memory have been implemented using Stratix III and Cyclone III FPGAs. The maximum clock speed that can be obtained by these packet classification engines when implemented using an FPGA is much slower than the maximum clock speed that can be obtained by an FPGA's internal memory. This is due to the logic delay in the components used by the engines such as the comparator blocks that compare a packet header with rules in a leaf node. It is therefore necessary to use multiple engines working in parallel so that the packet classification hardware accelerator can achieve maximum throughput. The use of multiple engines will help to ensure that the bandwidth of an FPGA's internal memory is better utilised.

Another reason for using multiple packet classification engines working in parallel is that it allows rulesets that contain many wildcard rules to be broken up into groups. The engines can work in parallel to classify a packet, with each group being searched using a separate engine. The matching rule with the highest priority (rule with the lowest rule ID) will be chosen in the case where multiple engines return a matching rule. The search structure for each group can be saved to the same block of memory that is shared by the engines. Splitting up rulesets that contain many wildcard rules into groups can help to reduce the amount of memory needed to save a ruleset's search structure and reduce the worst case number of memory accesses it takes to classify a packet, improving throughput. This is possible because rules where wildcard ranges occur in the same fields can be grouped together, with these fields not used for cutting, where possible. This makes it easier to divide a ruleset into sub-regions that contain a small number of rules and reduces the replicated storage of rules.

This section explains how the packet classification engines can be configured to work in parallel, sharing the same memory. It also explains the architecture of additional building blocks required to allow multiple engines to work in parallel. These building blocks include a high speed packet buffer used to capture the fields of a packet's header required for packet classification. The packet buffer is also used to distribute the packet headers among the classification engines. The classification results and packet ID for each engine is inputted into another building block known as a sorter logic block. This logic block has two functions. The first is to compare matching results between engines in the case where the ruleset has been split into multiple groups. This is done to make sure that the matching rule with the highest priority is selected in the case of multiple rule matches. The second function is to make sure that the classification results for the packets are outputted in the same order as the order that the packets were captured by the buffer.

3.7.1 Architecture of Packet Buffer

The packet buffer stores the source and destination IP address, source and destination port number and protocol number from the incoming packets at speeds of up to 250 Mpps. This allows the hardware accelerator to operate at line speeds

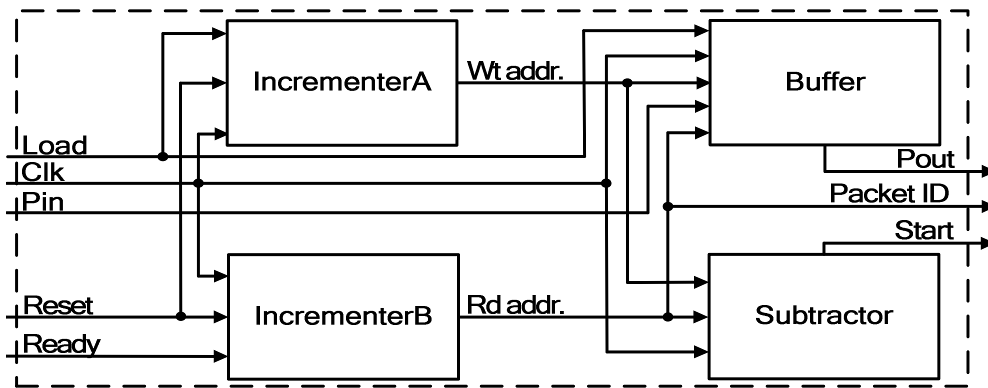


Fig. 3.28. Architecture of packet buffer used by packet classifiers.

in excess of 40 Gbps. The buffer works on a first come, first served basis, with packets being outputted from the buffer to the packet classification engines in the same order that they were inputted. The architecture of this buffer can be seen in Fig. 3.28. Every time a packet header appears at the input a *Load* signal will be asserted. This *Load* signal increments the write address that gives the memory location where the packet header will be saved in the buffer.

The packet classification engines as mentioned previously will assert a *Ready* signal when they are ready to classify a packet and there are packets to be classified. This signal will increment the read address of the buffer so that packet headers are read from the correct location. The write and read addresses of the packet buffer are subtracted from each other, with a difference between these addresses causing a *Start* signal to be asserted. The *Start* signal is used to notify the packet classification engines when there are packets ready to be classified. The read address is also outputted from the packet buffer with the packet header and used as a Packet ID. The Packet ID is used to make sure that the matching rule IDs are outputted by the hardware accelerator in the same order that the packet headers were inputted to the system.

3.7.2 Architecture of Sorter Logic Block

Fig. 3.29 shows the architecture of the sorter logic block used to make sure that the matching rule IDs are outputted in the correct order and that the rule with the highest priority is selected when there are multiple rule matches in the case where rulesets are broken up into groups. The sorter logic block accepts the *Match*, *No Match*, *Rule ID* and *Packet ID* signals from each of the packet classification

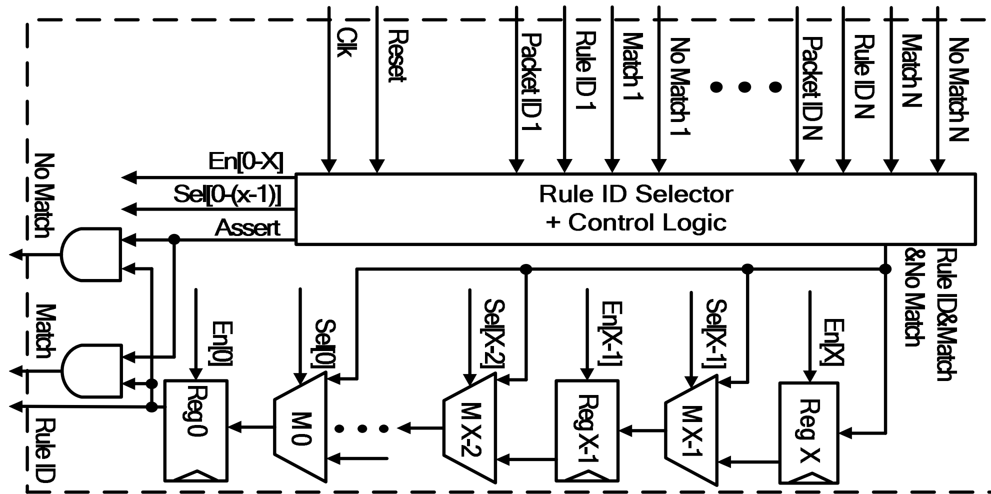


Fig. 3.29. Architecture of sorter logic block used by packet classifiers.

engines. It knows that an engine has finished classifying a particular packet represented by its *Packet ID* when either the *Match* or *No Match* signals have been asserted. The logic block labelled *Rule ID Selector + Control Logic* is used to make sure that the rule with the highest priority is selected between engines working in parallel to classify the same packet.

The *Rule ID Selector + Control Logic* block registers the *Match, No Match* and *Rule ID* signals for a packet that has been classified to a chain of registers and multiplexers in series. The register selected will depend on the *Packet ID* number. The *Match, No Match* and *Rule ID* will be registered to the output register if it is next in the sequence of packet results to be outputted, and stored if not. All stored rules will be shifted towards the output register each time a rule appears that is due to be outputted. This process is hidden, with the packet classification hardware accelerator outputting the result of classified packets on a first come, first served basis.

3.7.3 Architecture of Classifier Using Ultra-Wide Memory Words

The packet classification hardware accelerator designed to use ultra-wide memory words employs four packet classification engines working in parallel when implemented on a Stratix III FPGA and two engines working in parallel when implemented on the smaller low power Cyclone III FPGA. Both implementations use 7,704-bit wide memory words, with the Stratix III implementation having

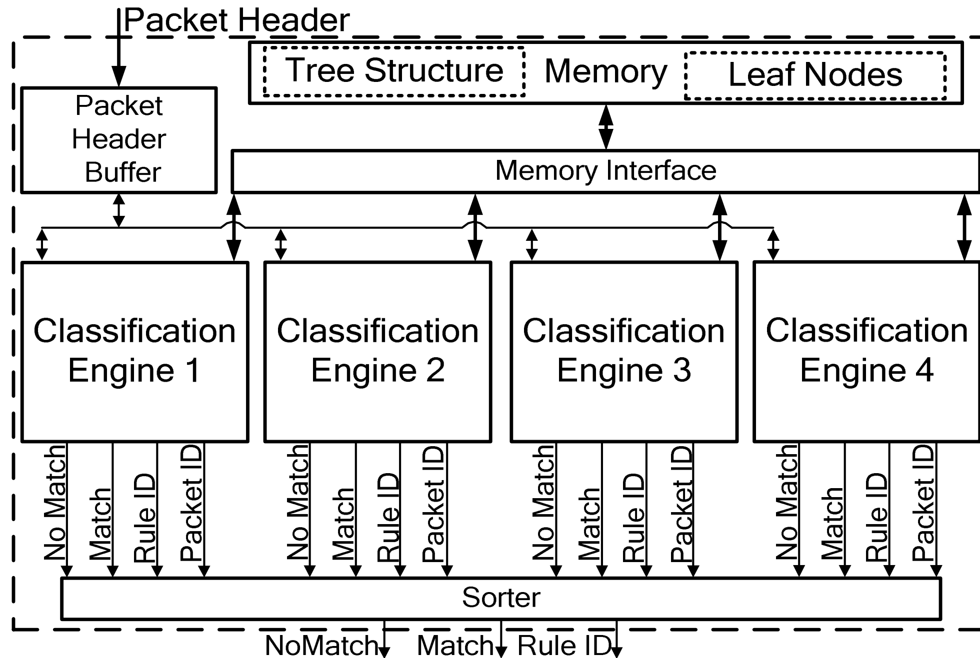


Fig. 3.30. Architecture of hardware accelerator using ultra-wide memory words.

1,024 memory words available to save the search structures required for classifying packets and the Cyclone III implementation having 512 memory words available.

Fig. 3.30 shows the architecture of the Stratix III implementation of the packet classification hardware accelerator. Its engines share access to the same memory port, with the four engines running at the same clock speed. Each engine uses a separate clock that is 90º out of phase with the previous engine. This is done to create a simple memory interface, with the read address of the four engines multiplexed together. This means that the memory must run at a speed equal to four times that of each engine to ensure that all engines are never refused a memory access. Each engine will therefore be guaranteed 25% of the available memory bandwidth. The sorter logic block must also run at the same clock speed used by the memory due to the fact that each engine is capable of classifying a new packet on every clock cycle when the decision tree only contains a root and leaf nodes, with a leaf node containing no more than 48 rules.

The *Ready* signals from the engines are also multiplexed together and inputted into the packet header buffer, with each engine having equal access to the packet buffer. The presence of four engines allows rulesets to be split into a maximum of four groups. Splitting a ruleset into four separate groups will mean that each

packet being classified requires the use of all four engines working in parallel to search through the four resulting search structures. This means that the maximum throughput for the hardware accelerator will be equal to that of an individual engine. Each packet will require two classification engines to find a matching rule in the case where the ruleset has been split into two separate groups as only two search structures need to be searched. This increases the maximum throughput of the hardware accelerator to twice the maximum throughput of a single classification engine. This is due to the fact that the hardware accelerator will contain two pairs of engines working in parallel. Maximum throughput can be obtained in the case where the ruleset does not need to be split into multiple groups. This is because there will only be one search structure, meaning that a single engine can classify a packet on its own. The maximum throughput for the hardware accelerator will therefore be equal to the sum of the throughput of all four engines working in parallel.

The architecture of the Cyclone III implementation of the packet classification hardware accelerator is almost identical to that of the architecture shown in Fig. 3.30. The only difference is that it uses two engines, which again run at the same clock speed, with the clock of each engine this time out of phase by 180º and the memory running at a clock speed twice that of an engine. Rulesets can only be split into a maximum of two groups when using this implementation as there are only two engines available to search through the resulting search structures.

3.7.4 Architecture of Classifier Using Reduced Width Memory Words

The 7,704-bit wide memory words used by the architecture explained in Section 3.7.3 has limited the Stratix III implementation to four packet classification engines and the Cyclone III implementation to two engines. This is due to limitations in the available logic interconnect within these devices. The architecture presented in this section, which uses on-chip reduced width memory, does not suffer from such limitations in available logic interconnect as it uses considerably smaller 324-bit wide memory words. It has also been implemented using a Stratix III and Cyclone III FPGA, with both implementations using eight packet classification engines. It takes advantage of the fact that the internal memory of an FPGA is dual port by placing two separate packet classifiers in

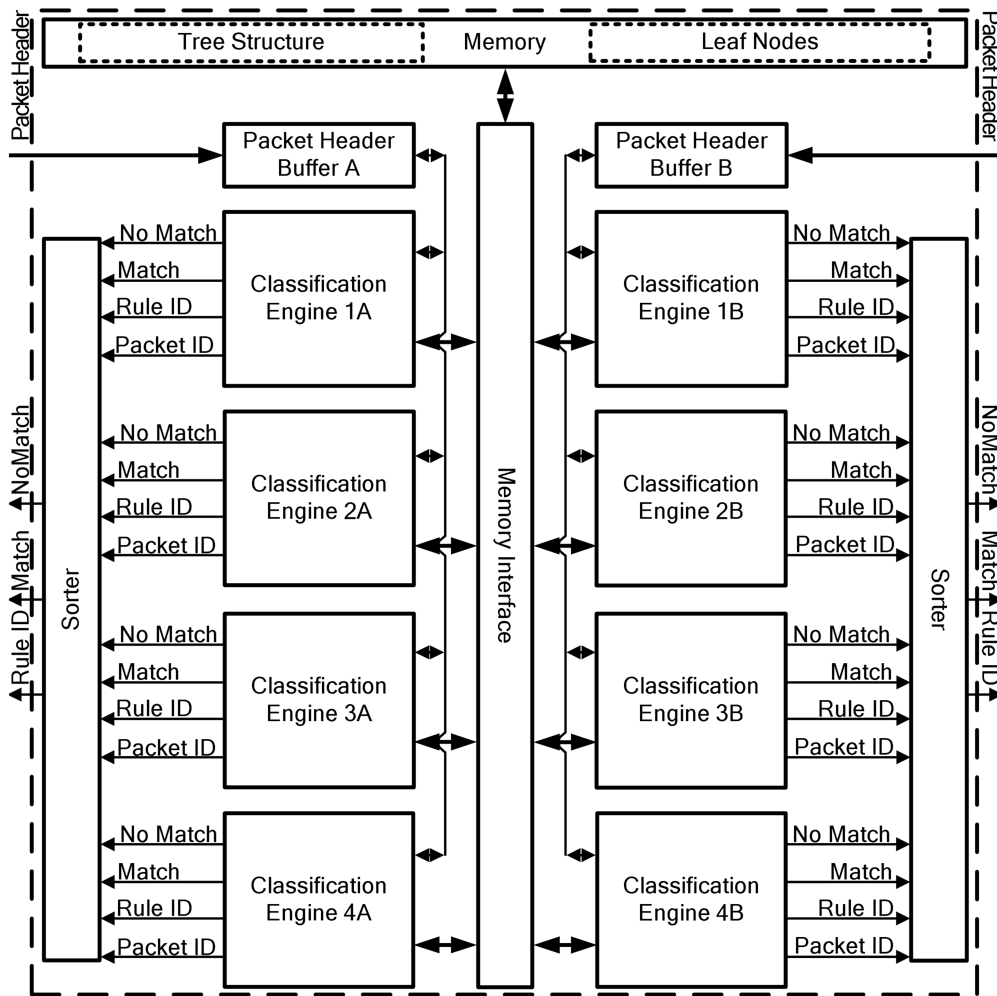


Fig. 3.31. Architecture of hardware accelerator using reduced width memory words.

parallel, sharing the same memory. Each classifier reads data from a separate data port and has its own packet buffer for storing the headers of incoming packets, four engines that work in parallel to maximise the bandwidth usage of a data port and a sorter logic block used to make sure that the classification results are outputted in the correct order.

The four engines belonging to a classifier again run at the same clock speed, with the clock used by each engine 90º out of phase with the clock used by the previous engine. Memory runs at a speed equal to four times that of an engine, ensuring a simple memory interface with each engine guaranteed access to memory on each of its clock cycles. Fig. 3.31 shows the hardware accelerator's architecture. The Stratix III implementation of this hardware accelerator has 46,080 memory words available to save the search structures required for

classifying packets, while the Cyclone III implementation has 12,288 memory words available. The memory used in each implementation is made up of a series of small memory blocks which are connected up so that they act as a continuous memory space. The memory ports of each memory block have their own enable signals. These enable signals are used to reduce power consumption by only activating the memory blocks that are being read from on a given clock cycle. This architecture also allows the splitting of a ruleset used to classify packets into groups of four or two in order to reduce the memory consumption and the worst case number of memory accesses needed to classify a packet for rulesets containing a large number of wildcard rules.

3.8 Performance Results

The hardware accelerator architectures designed to implement the modified HyperCuts packet classification algorithm have been tested extensively by measuring their logic and memory usage, throughput in terms of Mpps, amount of memory they require when storing the search structures needed to classify packets for the ACL, FW and IPC rulesets used in testing, worst case number of memory accesses required to classify a packet and power consumed when classifying packets. These results have been benchmarked against state of the art dedicated packet classification hardware accelerators where possible. The ACL, FW and IPC rulesets and their corresponding packet traces have been explained in Section 2.2. These rulesets and packet traces were generated using the ClassBench suite of tools.

3.8.1 Hardware Implementation Parameters

The packet classification hardware accelerator architectures presented in Section 3.7 were implemented in VHDL and targeted at two devices:

- A Cyclone EP3C120F484C7 FPGA, which is built on TSMC 65nm process technology, running at 1.2 Volts.
- A Stratix EP3SE260H780C2 FPGA, which is also built on TSMC 65nm process technology, running at 1.1 Volts.

The architectures were synthesised using Altera's Quartus II design software to obtain maximum clock speeds and resource utilisation summaries. The logic and

Table 3.3. FPGA resource utilisation for packet classification hardware accelerators.

Device	Logic element usage	Memory usage	f_{max}
Ultra-Wide Memory Words			
Cyclone III	45,244/119,088 (38%)	M9Ks 431/432 (99.8%)	65 MHz
Stratix III	121,797/254,400 (47.9%)	M9Ks 859/864,M144Ks 0/48 (52.6%)	169 MHz
Reduced Width Memory Words			
Cyclone III	23,491/119,088 (19.7%)	M9Ks 432/432 (100%)	219 MHz
Stratix III	40,070/254,400 (15.7%)	M9Ks 852/864,M144Ks 48/48 (99.3%)	433 MHz

memory usage of these architectures, along with the maximum clock speed that they can obtain are shown in Table 3.3.

It can be seen from looking at the table that the architecture that uses reduced width memory words is by far the best performer in terms of maximum achievable throughput and low logic usage. Its memory can achieve a maximum clock speed of 433 MHz when implemented using a Stratix III FPGA, giving it a maximum throughput of 433 Mpps. This is possible because each of its engines can classify a packet in two memory accesses and dual port memory is used, allowing two memory accesses to be made per clock cycle. A maximum throughput of 433 Mpps makes it the first packet classification hardware to the best of the author's knowledge that can process packets at line rates of up to 138.56 Gbps. To meet these line speeds the hardware accelerator needs to be able to process 433 Mpps as minimum sized 40 byte packets can arrive back-to-back. The Stratix III implementation of this architecture uses 99.3% of the FPGA's internal memory, allowing it to store the search structure required for rulesets containing in excess of 80,000 rules.

The Cyclone III implementation of this architecture also achieves a high throughput, with its memory obtaining a maximum clock speed of 219 MHz. This allows it to reach line speeds of up to 70 Gbps or 219 Mpps. The Cyclone III implementation uses 100% of the FPGA's internal memory, allowing it to store the search structure required for rulesets containing over 20,000 rules. These high levels of throughput make it possible for the Stratix III and Cyclone III implementations to easily cope with core network line speeds, which currently run at a maximum speed of 40 Gbps. These line speeds can be sustained by the classifier when it is used to classify packets for rulesets containing tens of thousands of rules.

The architecture that uses ultra-wide memory words also has no problems in coping with core network line speeds. The memory used in the Stratix III implementation of this architecture has a maximum clock speed of 169 MHz. This allows it to achieve a maximum throughput of 169 Mpps as it is possible for its engines to classify a packet in a single memory access. Its maximum achievable throughput is slower than that of the architecture that uses reduced width memory words for two reasons. The first is that its classification engines contain 48 comparator blocks, which are needed to compare a packet to the rule information returned from memory. This leaves them with a larger logic delay than is found in the engines that use reduced width memory words as they only require the use of 2 comparator blocks. The second reason is that the use of ultra-wide memory words only leaves enough logic interconnect for four engines to be used. The availability of more engines would allow dual port memory to be used, which could increase throughput by up to 100%.

The Stratix III implementation of the architecture with ultra-wide memory words uses 52.6% of the FPGA's internal memory, allowing it to store the search structures required to classify packets for rulesets containing up to 49,000 rules. The M144K block RAM is not used in the FPGA when implementing this architecture as it is not well suited to being configured as shallow memory with ultra-wide memory words. The Cyclone III implementation of the architecture uses 99.8% of its available memory resources, allowing it to store the search structure of rulesets containing up to 24,000 rules. The maximum clock speed that can be obtained by this memory is 65 MHz, allowing it to achieve a maximum throughput of 65 Mpps.

3.8.2 Memory Usage and Worst Case Number of Memory Accesses

The amount of memory required to save the ACL, FW and IPC search structures built for the packet classification hardware accelerator architectures using the modified HyperCuts algorithm can be seen in Table 3.4. This table also shows the worst case number of memory accesses required by the hardware accelerators to classify a packet when using these search structures. The results followed by an * show where a ruleset has been split into two groups in order to reduce the memory needed to save its search structure and to reduce the worst case number of

Table 3.4. Memory usage (bits) and worst case number of memory accesses.

Ruleset and number of rules	Ultra-Wide Memory Words				Reduced Width Memory Words					
	Stratix III		Cyclone III		DDR2 SDRAM		Stratix III		Cyclone III	
	Memory	MA	Memory	MA	Memory	MA	Memory	MA	Memory	MA
ACL75	23,112	1	23,112	1	48,096	2	23,004	2	23,004	2
ACL300	69,336	1	69,336	1	190,944	2	90,396	2	90,396	2
ACL1200	246,528	1	246,528	1	1,352,736	2	526,500	2	526,500	2
ACL2500	500,760	1	500,760	1	2,719,584	2	1,068,876	2	1,068,876	2
ACL5000	1,093,968	1	1,093,968	1	3,079,584	2	1,473,876	2	1,473,876	2
ACL10000	2,303,496	1	2,303,496	1	3,799,584	2	2,283,876	2	2,283,876	2
ACL15000	3,497,616	1	3,497,616	1	4,519,584	2	3,093,876	2	3,093,876	2
ACL20000	3,975,264	2	3,312,720	3	5,239,584	2	3,903,876	2	3,903,876	2
ACL24920	5,878,152	2			5,948,064	2	4,700,916	2		
FW75	23,112	1	23,112	1	29,664	2	17,820	2	17,820	2
FW300	69,336	1	69,336	1	190,944	2	90,396	2	90,396	2
FW1200	246,528	1	246,528	1	1,352,736	2	526,500	2	526,500	2
FW2500	516,168	1	516,168	1	1,539,936	2	753,624	2	753,624	2
FW5000	986,112	1	986,112	1	1,899,936	2	1,491,696	2	1,491,696	2
FW10000	4,707,144 1,440,648*	2 2*	3,798,072 1,440,648*	3 2*	24,976,800 4,979,520*	6 4*	7,968,456 2,615,976*	4 4*	3,933,360 2,615,976*	39 4*
FW15000	6,879,672 3,189,456*	3 2*	3,628,584 3,189,456*	8 2*	26,341,632 5,699,520*	6 4*	11,708,388 3,425,976*	4 4*	3,425,976*	4*
FW20000	7,311,096 3,782,664*	4 4*	3,898,224 3,782,664*	27 4*	74,170368 6,431,040*	7 4*	14,543,388 4,235,976*	16 4*	3,567,240	6*
FW23087	7,318,800 4,314,240*	7 4*	3,929,040*	5*	141,404,256 6,864,192*	7 4*	14,747,832 4,736,232*	53 4*	3,914,244*	9*
IPC75	46,224	1	46,224	1	48,096	2	23,044	2	23,044	2
IPC300	100,152	1	100,152	1	633,312	2	214,812	2	214,812	2
IPC1200	285,048	1	285,048	1	1,352,736	2	526,500	2	526,500	2
IPC2500	546,984	1	546,984	1	2,719,584	2	1,068,876	2	1,068,876	2
IPC5000	1,047,744	1	1,047,744	1	3,079,584	2	1,473,876	2	1,473,876	2
IPC10000	2,080,080	1	2,080,080	1	3,799,584	2	2,283,876	2	2,283,876	2
IPC15000	3,782,664	1	3,782,664	1	4,519,584	2	3,093,876	2	3,093,876	2
IPC20000	4,167,864	2	3,328,128	3	5,239,584	2	3,903,876	2	3,903,876	2
IPC24274	5,870,448	2			5,855,040	2	4,596,264	2		

memory accesses needed to classify a packet. The rulesets used for testing contain between 75 and 25,000 rules. This is more than enough rules to test the algorithm and hardware accelerator architectures, with research in [12, 36] showing that rulesets do not usually contain more than a thousand rules.

It can be seen that the amount of memory needed to save the search structures is the same for both the Cyclone and Stratix implementations of the packet classifiers when smaller rulesets are used. This is because the amount of available memory does not restrict how a decision tree is made. The amount of memory needed to save the search structures differs for larger rulesets because the Cyclone

III FPGA has less memory available which means it must build a deeper decision tree that uses less memory but takes extra memory accesses to classify a packet.

The results show that all architectures perform well in terms of memory consumption and worst case number of memory accesses when the ACL and IPC rulesets are used. The amount of memory needed to save the search structures built from the ACL and IPC rulesets is linear to the number of rules in the rulesets for all architectures, showing that the modified algorithm scales well to large rulesets. The architecture that uses ultra-wide memory words requires the least amount of memory on average because the fewest number of cuts are performed when building a decision tree. Few cuts are performed because leaf nodes can contain a large number of rules and do not therefore need to be broken into small sub-regions containing a few rules. The architecture that uses external reduced width memory requires the most memory to save the search structure built from a ruleset because of the large number of cuts that are made when building a decision tree and the large amount of information that it needs to store in a pointer. The architecture that uses reduced width memory words can achieve maximum throughput for all ACL and IPC rulesets tested, independent of whether internal or external memory is used, with a worst case number of 2 memory accesses needed to classify a packet. The architecture that uses ultra-wide memory words can achieve maximum throughput for the ACL and IPC rulesets containing up to 15,000 rules, with a slight drop off in throughput for larger rulesets. This is because a deeper decision tree will have to be traversed due to restrictions in the number of cuts that can be performed to an internal or root node.

The FW rulesets tested do not show the same high performance seen when using the ACL and IPC rulesets. This is because of the large number of wildcard rules that are contained within the FW rulesets. The architecture that uses on-chip reduced width memory, for example, requires 53 memory accesses at worst to classify a packet when using the search structure built for the FW ruleset containing 23,087 rules. This search structure requires 14,747,832 bits of memory to be saved. The architecture that uses the ultra-wide memory has been designed specifically to maintain high performance when rulesets that contain a large number of wildcard rules are used. This is because it can have leaf nodes that contain large numbers of rules due to the fact that it can search up to 48 rules in a

single memory access. It only requires 7 memory accesses to classify a packet when using the search structure built for the FW ruleset containing 23,087 rules, with this search structure requiring 7,318,800 bits of memory to be saved. This is a large improvement when compared to the architecture designed to use on-chip reduced width memory.

A worst case number of memory accesses of 53 for the architecture that uses on-chip reduced width memory and 7 for the architecture that uses on-chip ultra-wide memory will affect their performance by severely reducing their throughput. It will reduce the throughput of the classifier that uses reduced width memory words from its maximum of 433 Mpps to a worst case of 16.34 Mpps, while the classifier that uses ultra-wide memory words will have its throughput reduced from a maximum of 169 Mpps to a worst case of 24.143 Mpps. These throughputs can be increased by splitting the FW ruleset containing 23,087 rules into two different groups, with two packet classification engines used to classify each packet. Splitting the ruleset will mean that both classifiers will only require 4 memory accesses at worst to classify a packet, increasing the worst case throughput for these architectures to 216.5 and 42.25 Mpps respectively.

3.8.3 Throughput vs. Power Consumption

The power consumed by the packet classification hardware accelerators designed to use on-chip memory has been estimated by simulating them using ModelSim, with the packet headers generated by ClassBench used as input stimulus. The switching transitions on each node in a hardware accelerator were recorded for the duration of a simulation using a Value Change Dump (VCD) file. This VCD file was then analysed using the Quartus 2 PowerPlay Power Analyzer Tool to estimate the hardware accelerator's power consumption. The PowerPlay Power Analyzer Tool used post place and route information of the hardware accelerator when analysing the VCD files to accurately estimate the power consumption. The search structures used to estimate the power consumption in the results presented were created using the ACL ruleset containing 10,000 rules, with other rulesets showing similar results.

Fig. 3.32 shows the power consumed by the two packet classification hardware accelerator architectures implemented using the Cyclone III FPGA. This graph

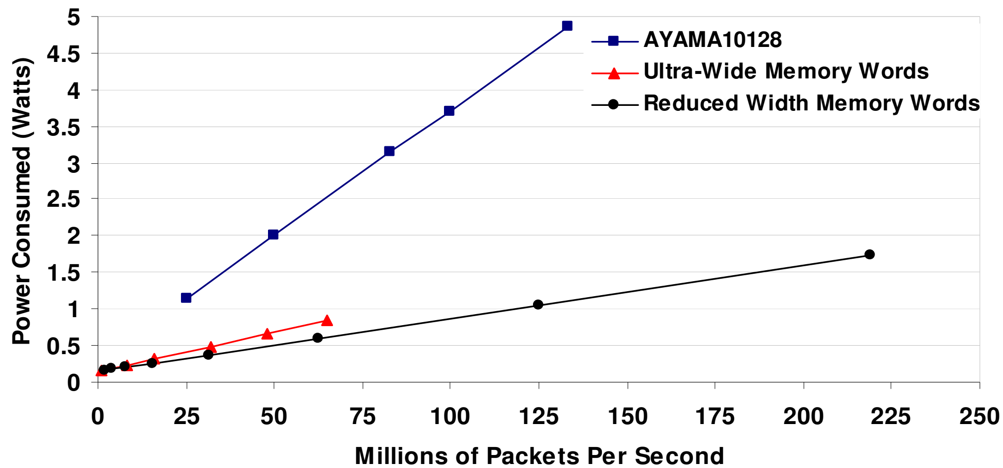


Fig. 3.32. Power consumed by packet classifiers implemented using Cyclone III.

was created by measuring the power consumed by a hardware accelerator while its clock speed and traffic volume were adjusted to different levels of throughput. It also shows the power consumed by the state of the art Cypress Ayama 10128 search engine [20], which classifies packets using TCAM. It contains a similar amount of memory to that used in the hardware accelerators implemented on the Cyclone III, allowing a fair comparison of power consumption and throughput to be made. The Cyclone III implementation of the hardware accelerator that uses ultra-wide memory words has 3,944,448 bits of memory available to save the search structures required to classify a packet. The implementation that uses reduced width memory words has 3,981,312 bits of memory available whilst the Cypress Ayama 10128 search engine has 4,608,000 bits available.

It can be seen that the hardware accelerator that uses ultra-wide memory words is the worst performer in terms of maximum throughput as it can classify 65 Mpps at best. Its peak power consumption at this level of throughput is 0.846 Watts which is similar to the 0.617 Watts consumed by the classifier that uses reduced width memory words, when classifying packets at the same speed. These power figures are low compared to the Cypress Ayama 10128 search engine which consumes 2.511 Watts when classifying 65 Mpps. The Cypress Ayama 10128 search engine has a maximum power consumption of 4.86 Watts when classifying packets at its maximum speed of 133 Mpps. This is high compared to the hardware accelerator that uses reduced width memory words as it only consumes 1.11 Watts when classifying packets at the same speed. The maximum power consumption of this

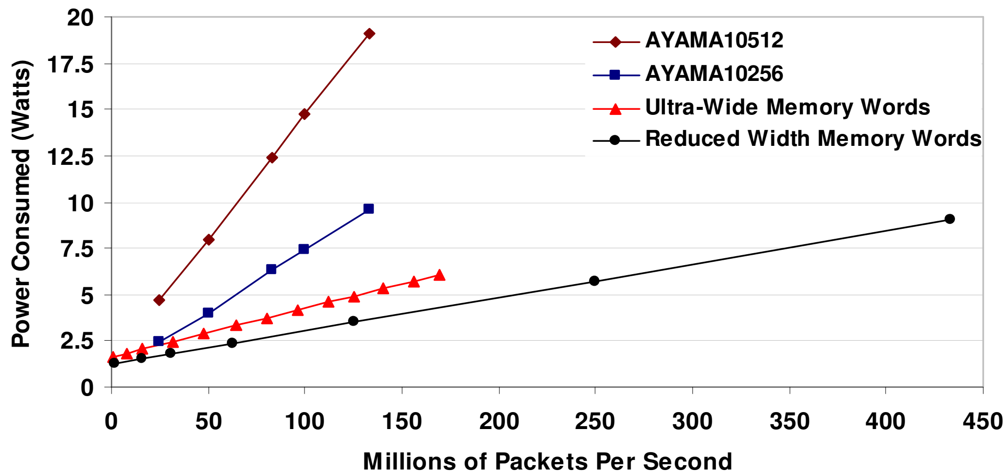


Fig. 3.33. Power consumed by packet classifiers implemented using Stratix III.

hardware accelerator is 1.73 Watts when classifying packets at its maximum throughput of 219 Mpps. This is a 65% performance increase compared to the maximum throughput that can be achieved by the state of the art packet classifier that uses TCAM. This is impressive when the massive reductions in power consumption of up to 77% that have also been achieved are considered.

The Stratix III implementations of the packet classification hardware accelerators can achieve an even greater performance increase in terms of maximum achievable throughput when compared to the state of the art packet classifiers that use TCAM. Fig. 3.33 shows the power consumed by the two packet classification hardware accelerator architectures implemented using the Stratix III FPGA. It also shows the power consumed by the state of the art Cypress Ayama 10256 and 10512 search engines. The Cypress Ayama 10256 search engine has 9,216,000 bits of memory available to save the search structure needed to match packets to the rules in a ruleset. This means that it can be compared to the classifier that uses ultra-wide memory words with 7,888,896 bits of memory available. The Cypress Ayama 10512 search engine has 18,432,000 bits of memory available, making it suitable for comparison with the classifier that uses reduced width memory words with 14,929,920 bits of memory available.

It can be seen that the Cypress Ayama search engines are the slowest, with a maximum throughput of 133 Mpps. At this speed the Cypress Ayama 10256 search engine consumes 9.57 Watts, while the classifier that uses ultra-wide memory words only consumes 5.12 Watts, even though it has a similar amount of

memory. The Cypress Ayama 10512 search engine consumes 19.14 Watts when classifying 133 Mpps, with the classifier that uses reduced width memory words consuming 3.64 Watts when classifying packets at the same speed. The maximum power consumption for the classifiers that use ultra-wide and reduced width memory words is 6.08 and 9.03 Watts respectively when classifying packets at their top speeds of 169 and 433 Mpps. It can be seen that the power consumption is much higher for the Stratix III implementations compared to the Cyclone III implementations. This is because the Stratix III is a much larger device, with greater amounts of logic and memory resources available, leading to a larger amount of static power consumption. The larger amount of memory and logic used in the Stratix III implementations combined with the higher speeds will also cause more dynamic power consumption due to an increased amount of switching.

3.8.4 Evaluation Against Prior Art

The area of packet classification is a well studied field. Most research, however, has concentrated on the implementation of new packet classification algorithms tailored towards increased performance with software implementation in mind. These algorithms rarely consider the effects of power consumption, with their main aims instead being to increase the storage efficiency of rulesets while reducing the number of memory accesses needed to classify a packet. Section 2.3 explains several such algorithms.

Research into the improvement of packet classification algorithms for increased throughput through hardware acceleration with reduced power consumption remains limited. This is an increasingly important field of research as hardware accelerators have become essential when trying to meet network line speeds, which are growing steadily due to advances in optical fibre technology. Performing packet classification at these ever-increasing line speeds is made more difficult by the fact that rulesets are expanding because of the ever-increasing number of services that need to be provided. Most state of the art packet classification hardware accelerators aim to increase throughput through the use of TCAM [60, 61, 62, 81, 82, 83, 84, 85, 86]. The use of TCAM, however, makes these approaches a power hungry solution, even if power reduction techniques are used.

Table 3.5. Performance comparison of packet classification hardware accelerators.

Approach	Device	ACL Rules	Throug-hput	Memory Usage (bits)	Logic usage (6-LUT)
Pipelined Tree [88]	Virtex 5	9,603	250 Mpps	5,013,504	41,228/122,880
Ultra-Wide Memory	Stratix III	10,000	169 Mpps	2,303,496	48,719/101,760
Reduced Width Memory	Stratix III	10,000	433 Mpps	2,283,876	16,028/101,760

Packet classification hardware accelerators targeted towards the use of FPGAs and memories such as SRAM instead of high power TCAM include the work presented in [87]. It introduces a packet classification algorithm known as Distributed Crossproducting of Field Labels. The algorithm uses multiple search engines that work in parallel, with a separate search engine used to match each field of a packet's header to the corresponding field of the rules within a ruleset. Each engine will return the rules that matched the field it searched. An aggregator looks at the matching results from each field and picks the rules where all fields within a rule match the packet header. The matching rules are passed to a priority resolution stage that picks the rule with the highest priority as the matching rule. The search engines used are tailored towards the fields that they search. The engines that search the source and destination IP addresses are tailored to perform prefix matching, the source and destination port numbers use engines tailored towards range matching, while the protocol number uses an engine suited to performing exact matching. The authors claim that their architecture could classify 100 million packets per second while using rulesets containing up to 200 thousand rules. These performance figures are unlikely, however, as they assume that their logic intensive architecture could run at the maximum clock frequency of an FPGA.

Table 3.5 compares the performance of the Stratix III implementation of the packet classification hardware accelerators presented here against another packet classifier based on the HyperCuts algorithm. The work in [88] implements a decision tree-based, dual pipeline architecture that can classify 250 Mpps when using rulesets containing up to 10,000 rules. It proposes optimisation techniques to the HyperCuts algorithm such as a precise range cutting heuristic that reduces the replicated storage of rules. It also employs a tree to pipeline mapping scheme to improve memory utilisation. Drawbacks with this design include poor storage efficiency for rulesets containing many wildcard rules, meaning that very large

rulesets cannot be supported. Another drawback is that the architecture must be reconfigured if the depth of the decision tree constructed exceeds the worst case depth allowed by the implemented architecture, reducing the flexibility of the design and limiting it to FPGA implementations.

The performance of the hardware accelerators are compared when classifying packets using an ACL ruleset with 10,000 rules, generated using ClassBench. The performance metrics examined are their throughput in terms of worst case number of packets that they can classify per second, amount of memory needed to save the search structure required to classify packets and their logic usage. Power consumption is not compared as these results were never given by the authors of the architecture that uses a pipelined decision tree. The hardware accelerators presented here were implemented on a Stratix EP3SE260 FPGA, while the approach employed in [88] used a Virtex XC5VFX200T FPGA. A direct comparison is fair as the performance of both FPGAs is similar due to the fact that both are manufactured using 65nm process technology. Both devices also have similar amounts of internal memory resources available, with the Virtex having 16,809,984 bits of memory available and the Stratix having 15,040,512. The amount of logic available on both devices is also very similar. The logic usage of the hardware accelerators have been compared using 6 input Lookup Tables (LUT) as the Virtex gives the logic utilisation in slices, with each slice capable of implementing four 6 input LUT. The Stratix gives memory utilisation in adaptive logic modules, with each capable of implementing one 6 input LUT.

It can be seen that the hardware accelerator architectures presented here are by far the best performers in terms of the amount of memory needed to save the search structure created from an ACL ruleset. They use less than 50% of the memory required by the architecture that uses a pipelined decision tree. The architectures presented here also have the ability to show even higher reductions in memory consumption when using rulesets that contain many wildcard rules. They do this by breaking these problem rulesets into multiple sets of rules, which can be searched in parallel. The architecture that uses a pipelined decision tree cannot do this, meaning that it would struggle to scale to rulesets containing tens of thousands of rules. The architecture presented here, which uses the reduced width memory words, is by far the best performer in terms of throughput, classifying

nearly twice as many packets per second as the architecture that uses a pipelined decision tree. It also has a much smaller logic footprint, with its implementation requiring over 60% less 6 input LUT, compared to the architecture that uses a pipelined decision tree. This small logic footprint helps the architecture presented here to obtain higher clocking speeds. The architecture that uses the ultra-wide memory words has the largest logic footprint and lowest throughput when it comes to classifying packets using the ACL ruleset. This is because the architecture has been designed to maintain high performance on rulesets that contain many wildcard rules, such as a firewall ruleset, with the ACL ruleset only containing a small number of wildcard rules.

3.9 Summary of Contributions

This chapter has presented modifications to the HyperCuts packet classification algorithm that make it better suited to hardware implementation. These modifications include changing the cutting scheme so that the need for slow and logic intensive floating point division is removed when classifying a packet. This is done by replacing the region compaction scheme used by HyperCuts with a new scheme that uses pre-cutting. Pre-cutting reduces the number of sub-regions that need to be stored in a decision tree, thus reducing memory consumption. It also has the advantage of only requiring simple shift and AND operations to be used when calculating the path a packet should follow when traversing a decision tree. This simplifies the architecture of the hardware required to classify a packet, allowing increased clock speeds. Modifications are also made to how rules are stored through simple encoding schemes that improve the storage efficiency of rulesets.

Three new multi-engine packet classification hardware accelerator architectures were also presented that implement the modified HyperCuts algorithm. All three architectures can classify packets at core network line speeds using rulesets containing tens of thousands of rules. One of these architectures uses on-chip ultra-wide FPGA memory and is ideally suited to classifying packets using rulesets that contain many wildcard rules. Decision trees built from such rulesets tend to contain large leaf nodes due to the replicated storage of rules. The ultra-wide memory words make this architecture ideally suited to searching such

decision trees as it can search up to 48 rules of a leaf node in one memory access. It has a maximum throughput of 169 mpps. The remaining two architectures use reduced width memory words and are ideally suited to classifying packets for rulesets that do not contain a lot of wildcard rules. One of these architectures uses on-chip FPGA memory. The use of reduced width memory words and on-chip memory makes it possible for this architecture to classify up to 433 Mpps. The second architecture that uses reduced width memory words employs external memory, giving it the ability to classify packets when using rulesets containing up to one million rules.

Chapter 4 - Frequency Scaling Architecture

4.1 Introduction

The packet classification hardware accelerator architectures presented in Chapter 3 have been designed to achieve maximum throughput. They obtain a high throughput, while achieving low power consumption, when compared to other state of the art hardware-based platforms used to implement packet classification such as TCAM. They have not, however, been designed to implement power saving techniques that exploit the fact that a classifier does not always need to operate at its full processing capacity. This is because networks can experience large fluctuations in traffic, leaving room for a reduction in power consumption. A classifier may be kept busier during peak traffic times such as office hours in comparison to other times such as the night or during public holidays. At a micro level traffic can also fluctuate from second to second, with large peaks and troughs in throughput. This fluctuation in traffic means that it makes sense to reduce power consumption by adjusting the processing capacity of a classifier so that it is just enough to meet the processing needs of the network traffic. Matching the available processing capacity to the traffic volume will reduce the dynamic power caused by unnecessary switching.

This chapter presents a low power architecture for packet classification that uses an Adaptive Clocking Unit (ACU) to dynamically adjust the clock frequency of a packet classifier so that its processing capacity is just enough to meet the processing needs of a network's traffic. The chapter is laid out as follows. Section 4.2 presents findings of an analysis carried out on the throughput characteristics of real network traffic. It also shows the amount of time a classification engine spends idle when processing packets from a real packet trace, showing why the

low power architecture needs only one engine to meet line speeds up to 40 Gbps. A summary of techniques that can be used to reduce power consumption are given in Section 4.3. Section 4.4 presents the adaptive clocking scheme, explaining the methods employed to keep frequency switches to a minimum. It also explains the architecture of the ACU. The complete low power architecture for packet classification is explained in Section 4.5, along with the parameters of the hardware used to implement it. This section also presents the low power architecture's power usage figures. Section 4.6 shows the energy savings that can be made when the low power architecture is used to classify packets from synthetic traces running at line speeds of up to 40 Gbps. The chapter is summarised in Section 4.7.

4.2 Analysis of Real Traces

The Internet backbone is made up of a large collection of interconnected commercial and non-commercial high speed data links that are connected by edge and core routers. In the past 2.5 Gbps (OC-48) connections were used as the backbones by many regional ISPs. These connections can transmit a maximum of 7.8125 Mpps when the back-to-back arrival of minimum sized 40 byte packets is considered. Currently the most common commercial network connection speed is 10 Gbps (OC-192), which allows for a maximum throughput of 31.25 Mpps. With companies like AT&T already using 40 Gbps (OC-768) line speeds [89], it is envisaged that these high speed connections will become more commonly available in the near future. Line speeds of 40 Gbps can transmit a maximum of 125 Mpps. Any low power architecture for packet classification should therefore be designed so that it is able to meet these 40 Gbps line speeds.

A detailed analysis was carried out on the characteristics of real 2.5 and 10 Gbps traffic traces stored in a database belonging to the National Laboratory for Applied Network Research (NLANR) [90]. Traffic traces with throughputs of 40 Gbps could not be analysed as they have not yet become publicly available. The throughput of these traces was looked at in terms of both bits and packets per second. This was done because packet classifiers are more interested in throughput in terms of packets per second rather than bits per second, which is the metric most networking equipment is interested in. Classifiers are only interested

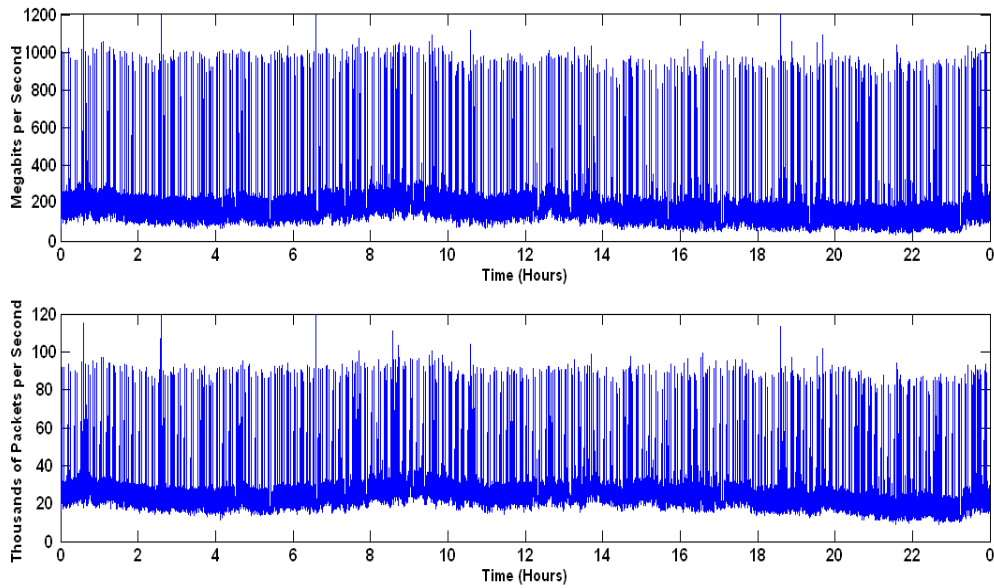


Fig. 4.1. Throughput of a 24-hour trace from the CENIC HPR backbone link.

Table 4.1. Statistics on packet sizes in the CENIC HPR backbone trace.

Number of Packets	Average Packet Length	Packet Length Distribution		
		0-200	201 -1400	1401-1500
2,607,169,713	975 bytes	33.56%	7.03%	59.41%

in throughput in terms of packets per second because they only examine a packet's header and not its payload.

Fig. 4.1 shows a 24 hour recording taken from the Corporation for Education Network Initiatives in California (CENIC) High Performance Research (HPR) backbone link [91]. The characteristics of this trace are typical of all the backbone traces that have been analysed, with the traffic load varying constantly, with many short bursts in throughput. It can be seen that these short bursts cause the throughput to fluctuate wildly from second to second both in terms of bits and packets per second. The trace shows that even during sharp bursts in throughput, the 10 Gigabit CENIC backbone link peaks at only 121,801 pps and never reaches its theoretically highest throughput of 31.25 Mpps. This is due to the fact that a large number of packets are sent across the network at the size of the Maximum Transmission Unit (MTU), which is 1,500 bytes at the network layer. This explains the large average packet size of 975 bytes. A breakdown of the packet length distribution can be seen in Table 4.1. Analysis of packet traces taken from the Sprint IP backbone network [92] and the ARIGE and UNINA Wide Area

Networks [93] show similar results, with large average sized packets due to packets being sent at the MTU.

4.2.1 Processing Needs

It is clear from the analysis of real traces that multiple packet classification engines are not needed to classify packets when looking at traffic volume in terms of packets per second. This is because a peak throughput of more than 1 Mpps is not obtainable for 10 Gbps connections due to the large packet sizes. Connections reaching speeds of 40 Gbps would therefore not be expected to reach throughputs of more than a few Mpps. The packet classification engines presented in Chapter 3 can easily cope with these levels of throughput. The engine presented in Section 3.6.1 that employs on-chip ultra-wide memory can classify up to 68 Mpps when implemented on its own, while the engine that employs on-chip reduced width memory can classify up to 62.5 Mpps.

The low power architecture for packet classification employs the engine that uses ultra-wide memory words because it performs best when there is only one engine available. Having only one engine available does not allow the option of breaking rulesets that contain many wildcard rules into groups, with each group searched in parallel using a separate engine. The engine that uses ultra-wide memory words performs best on rulesets that contain many wildcard rules because it can access large amounts of data in a single clock cycle. This allows it to quickly search through the large leaf nodes that occur in the decision trees built for rulesets that contain many wildcard rules. These leaf nodes are large due to the replicated storage of rules.

Section 3.8.2 shows that the FW ruleset with 23,087 rules is the hardest ruleset to classify packets for when there is only one engine available. This is because it is the ruleset with the largest number of wildcard rules. The engine that uses ultra-wide memory words requires 7 memory accesses at worst to classify packets when using this ruleset. Running the engine at a speed of 32 MHz would give it plenty of processing capacity to classify packets on a 40 Gbps connection. This would allow the engine to classify packets at a sustained rate of 4.5 Mpps, even if all packets needed a worst case of 7 memory accesses to be classified. The engine that uses reduced width memory words requires a worst case of 53 memory

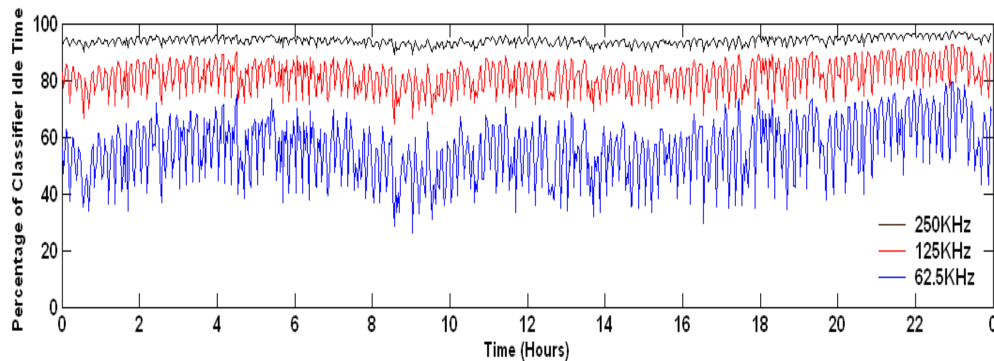


Fig. 4.2. Percentage of classifier idle time when classifying packets from the CENIC trace.

accesses to classify a packet when using the same rulesets. This would limit its worst case throughput at 32 MHz to 600 thousand packets per second, which would not be sufficient to meet 40 Gbps line speeds.

4.2.2 Classifier Utilisation

A cycle accurate simulator was developed in C code that contains a high speed buffer used to capture the fields from a packet header required for classification. It also includes a packet classification engine that uses ultra-wide memory words. The simulator was used to verify that a single engine could classify packets from the backbone traces stored in the NLANR database without dropping any packets, when using the ACL, FW and IPC rulesets used in testing. This input stimulus to the simulator was generated by splicing the timestamp from the packet headers in the NLANR traces to the packet headers generated by ClassBench for the test rulesets. It was found that the classifier could easily cope with all traces tested, with no packets dropped even when the classifier was run at a fraction of its maximum clock speed.

Fig. 4.2 shows the percentage of time the packet classifier spends in an idle state when classifying packets from the daylong, 10 Gigabit, CENIC HPR backbone trace shown in Fig. 4.1. The classifier was run at fixed clock speeds of 250 KHz, 125 KHz and 62.5 KHz. These speeds are well under the classifier’s maximum clock frequency of 68 MHz. The ruleset used when measuring the idle time for this graph was the ACL ruleset with 10,000 rules. Its search structure requires one memory access at worst to classify a packet. It can be seen that the classifier spends over 90% of its time in an idle state classifying no packets when run at 250 KHz. Running the classifier at a clock speed where its processing capacity is high

enough to cope with traffic volumes well above average will result in a small packet buffer being required. This is because packets will only have to spend a short time queuing before they are classified. A large amount of available processing capacity and high percentage of idle time will come at the expense of a large amount of dynamic power being needlessly wasted due to unnecessary switching in the clock tree. It has been estimated that the clock tree alone can account for 30-50% of the total power consumption in a digital circuit [94, 95].

It can be seen that the classifier's idle time drops to around 50% as its clock frequency is decreased to 62.5 KHz, reducing the amount of dynamic power wasted due to unnecessary switching. It is not, however, an ideal solution to run the classifier at a clock speed where its available processing capacity is only high enough to meet traffic volumes just above the average level. This is because a large packet buffer would be required to prevent packets being dropped during high bursts of traffic. The power used by a large high speed packet buffer would be more than the dynamic power saved in the classifier. A large buffer and slow classifier would also result in an unacceptably large delay in the amount of time it takes to process a packet. It is therefore clear that a method is needed to match the classifier's available processing capacity to the traffic volume so that dynamic power is reduced during times of low traffic and so that only a small packet buffer is required to cope with high bursts of traffic.

4.3 Methods for Reducing Power Consumption

There have been many methods proposed that aim to reduce the power consumed by devices that are capable of carrying out packet classification, such as programmable multi-core network processors. The proposed methods are used to adjust the available processing capacity of a network processor so that it matches the processing needs of network traffic over time. This section explains the most effective methods, stating which are well suited to reducing the power consumed by a dedicated packet classification hardware accelerator, and which are not.

4.3.1 Clock Gating/Turning Off Unused Processing Elements

One of the most popular methods used to reduce power consumption in digital circuitry is clock gating [96, 97]. Clock gating can be used when there are multiple Processing Elements (PEs) available to process data. It involves

switching off the clock to unneeded PEs at times when the workload is low. This reduces the dynamic power consumed in the unneeded PEs to almost zero as the unnecessary switching of logic elements is eliminated. Clock gating effectively turns off unused PEs without powering them down, which reduces the amount of time it takes to reactivate them when they are again needed. Leaving them powered up will, however, mean that static power consumption remains due to current leakage. Clock gating is used to turn off the clock of the unneeded PEs of multi-core network processors in [98] and [99] to reduce dynamic power consumption at times when there is low traffic volume. This results in energy savings of up to 30% and 40% respectively, with only a small drop in throughput.

The more aggressive approach of turning a network processor's PEs on and off using a power management controller is taken in [100], with the available processing capacity matched to the processing needs of the network traffic. It was found that this can reduce a network processor's core power consumption by 50-60%, with both static and dynamic power reduced. The disadvantage with completely turning off unneeded PEs is the large amount of time it takes to turn them back on. The large power savings of up to 60% come at the price of a large processing delay, with 50% of packets delayed by more than 600 μs.

The approaches of clock gating and turning off unneeded PEs are not used by the low power architecture for packet classification presented in this chapter. This is because the analysis of real network packet traces in the previous two sections found that one packet classification engine would be more than enough to process packets at line speeds of up to 40 Gbps, when even the most difficult rulesets are used. Turning off the only available classification engine during times of low traffic volume would result in unacceptably large processing delays and could even lead to packets being dropped if a large enough buffer was not used.

4.3.2 Voltage/Frequency Scaling

Another method that can be used to reduce the power consumed by electronic devices is to dynamically scale their clock frequency and/or supply voltage, reducing both dynamic and static power consumption [101, 102, 103, 104, 105, 106, 107]. This is a popular method of reducing power as it does not matter how many PEs a device has available to process data. An advantage of scaling the

clock frequency and voltage of a PE is that it never needs to be turned off. This means that there is a reduced delay when it comes to increasing and decreasing its available processing capacity. Frequency scaling reduces the dynamic power usage of a PE by reducing the unnecessary switching of logic elements, while reducing the supply voltage of a PE will reduce both its dynamic and static power consumption. Reducing the supply voltage of a transistor has an adverse effect on its speed, slowing it down. This means that a PE's supply voltage must be high when it is being run at a fast clock frequency during times when it has a heavy workload. The supply voltage and clock frequency can be reduced when the workload decreases, therefore saving power.

Dynamic voltage and frequency scaling were used to reduce the power consumption of a multi-core network processor in [108], leading to power savings of up to 17%, with throughput dropping by less than 6%. The packet classifier presented here does not use dynamic voltage scaling. This is because it has been implemented in testing using commercial FPGAs, where it is hard to implement dynamic voltage scaling as external circuitry is needed to control the voltage level [109]. The packet classifier presented here instead uses dynamic frequency scaling as it can be implemented when using either an FPGA or ASIC. Frequency scaling is also ideally suited to devices that have only one PE. The packet classifier uses an ACU that has been designed to dynamically scale the clock frequency of the packet classification engine and its memory so that its processing capacity matches fluctuations in traffic volume. It is possible to reduce the packet classifier's dynamic power consumption by running it at low speeds when traffic volume is low. It is also possible to reduce the buffer size, and therefore its power consumption, by allowing the packet classifier to respond to bursts of packets by increasing its clock frequency in order to keep the buffer clear.

4.4 Adaptive Clocking Scheme

The ACU employed by the packet classifier uses dual port SRAM to buffer information from the incoming packet headers. This information includes the header's source and destination IP addresses, source and destination port numbers and the protocol number, which are read in at a speed of 128 MHz. This speed is selected to avoid packets being dropped when the arrival of back-to-back 40 byte

Table 4.2. Clock speed associated with each state.

State	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
Speed MHz	f_0=0.0625	f_1=0.125	f_2=0.25	f_3=0.5	f_4=1	f_5=2	f_6=4	f_7=8	f_8=16	f_9=32

packets occurs at 40 Gbps line speeds, resulting in up to 125 Mpps as mentioned previously. The number of packets stored in the buffer is calculated by monitoring the difference between the buffer's read and write addresses. This difference is used as a trigger to determine which clock frequency the packet classification engine and its memory should be run at. The ACU has been designed to run the packet classifier at up to N different clock frequencies, with N being equal to 10 in the experimentation carried out here. Each of the N clocks are generated using a separate Phase Lock Loop (PLL) output clock. Devices such as Altera's Stratix III FPGAs contain up to 12 PLLs, with each PLL capable of generating 10 clocks, which can be configured to run at different frequencies. Smaller devices such as Altera's Cyclone III FPGAs contain up to 4 PLLs, with each PLL capable of generating 5 clocks. Each clock is generated using a separate PLL output clock to eliminate the need of PLL frequency changing that requires some time to finish. Dedicated clock switching logic in the FPGA and ASIC are used to prevent clock glitches when switching between frequencies. The packet classification engine is put into an idle state before switching clock frequencies to prevent problems that may occur due to clock glitches.

4.4.1 Method for Reducing Frequency Switching

The ACU can be easily modified to run the packet classifier at a wide range of clock frequencies. The clock frequencies selected to run the packet classifier here were calculated using Equation 4.1, where *Fmax* is the maximum clock frequency that the packet classifier is allowed to run at. This *Fmax* limit has been capped at 32 MHz even though the packet classifier could run at clock speeds of up to 68 MHz. The maximum clock frequency has been capped, as explained previously, due to the fact that the packet classifier has been designed to keep power consumption as low as possible and because 32 MHz is fast enough for the packet classifier to easily cope with 40 Gbps line speeds.

$$f_i = Fmax/2^{N-i-1}, \quad i=0, \ldots, N-1 \qquad (4.1)$$

The ACU uses up to N different states, with each state corresponding to a different clock frequency. Table 4.2 shows the clock frequencies associated with

each of the ten states for the experimentation carried out here. The entering and exiting of each state is triggered by the number of packets stored in the buffer. All states apart from state S_{N-1} have a threshold for determining how many packets can be stored in the buffer before the next higher frequency is used. These thresholds are variable, with the number of packets stored in the buffer distributed among the N states, with each state having a width W_i. The width of each state W_i can be any number between zero and M (total number of packets the buffer can store) as long as the Equation 4.2 is satisfied.

$$M = \sum_{i=0}^{N-1} W_i \qquad (4.2)$$

The ACU has been designed to be as flexible as possible. It allows the thresholds used to determine when a state is exited and the next higher state entered to be changed at any time. These thresholds are written to registers within the ACU. The threshold for each state can be calculated using Equation 4.3.

$$T_i = \sum_{j=0}^{i} W_j, \quad i=0, \ldots, N-2 \qquad (4.3)$$

The output clock frequency of the ACU always starts at the frequency of the lowest-used state and only changes to the frequency of the next higher-used state if the number of packets stored in the buffer exceeds its threshold. There are two conditions for leaving the used states between the lowest-used and highest-used states and thus changing the output clock frequency. The first of these conditions is that the number of packets in the buffer exceeds the threshold T_i for the current state S_i, with the output clock frequency scaling up to the next higher-used frequency. The second condition is that the number of packets stored in the buffer reaches zero, meaning that the output clock frequency scales down to the frequency of the lowest-used state. The highest-used state will only be exited and the output clock frequency changed if the buffer is cleared, changing the frequency to that of the lowest-used state. This means that the number of buffer slots that the current state can occupy before a frequency change is equal to the sum of the buffer slots occupied by the previous states plus the number of slots assigned to the current state itself. This is done to allow larger fluctuations in the number of packets stored in the buffer without unnecessary frequency drops. It

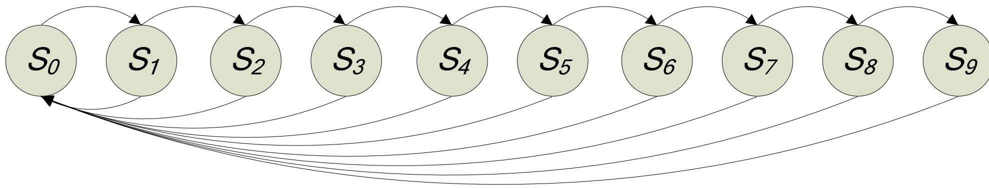


Fig. 4.3. Switching sequences with all states used.

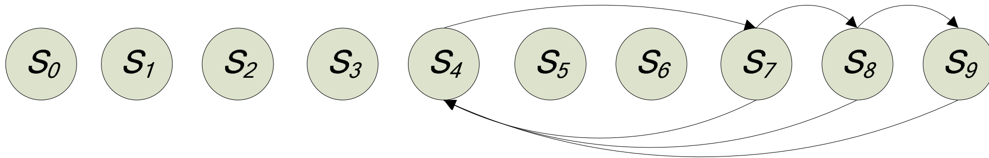


Fig. 4.4. Switching sequences with selected states used.

also keeps the latency time of processing a packet to a minimum by trying to clear the buffer before reducing the clock frequency. The clock frequency of the packet classifier remains fixed if all buffer slots are occupied by one state.

Fig. 4.3 shows an example where the buffer's slots are distributed equally among all states. In this example the output clock frequency to the ACU will start at f_0, the frequency of the lowest-used state S_0. If the threshold for this state T_0 is exceeded (i.e. the buffer slots assigned to state S_0 have been filled) then the next higher-used state S_1 will be entered and the clock frequency will change to f_1. The output clock frequency will remain at f_1 until the number of packets stored in the buffer is reduced to zero, returning the output clock frequency to f_0, or the threshold T_1 is exceeded in which case the output clock frequency changes to f_2. The same procedure is followed for all states between the lowest and highest used states. The output clock frequency will remain at f_2, for example, until either all packets in the buffer are cleared, returning the output clock frequency to f_0, or the maximum threshold T_2 is exceeded, meaning state S_3 is entered, with the output clock frequency changed to f_3. State S_9 can only be exited with state S_0 entered if all packets in the buffer are cleared.

Fig. 4.4 shows an example where only states S_4, S_7, S_8 and S_9 are used. In this case the output clock frequency to the packet classifier will start at f_4. It will stay at f_4 until the threshold T_4 is exceeded, increasing the clock frequency to f_7. The output clock frequency will stay at f_7 until all packets in the buffer have been cleared, returning the output frequency to f_4, or the threshold T_7 is exceeded, increasing the

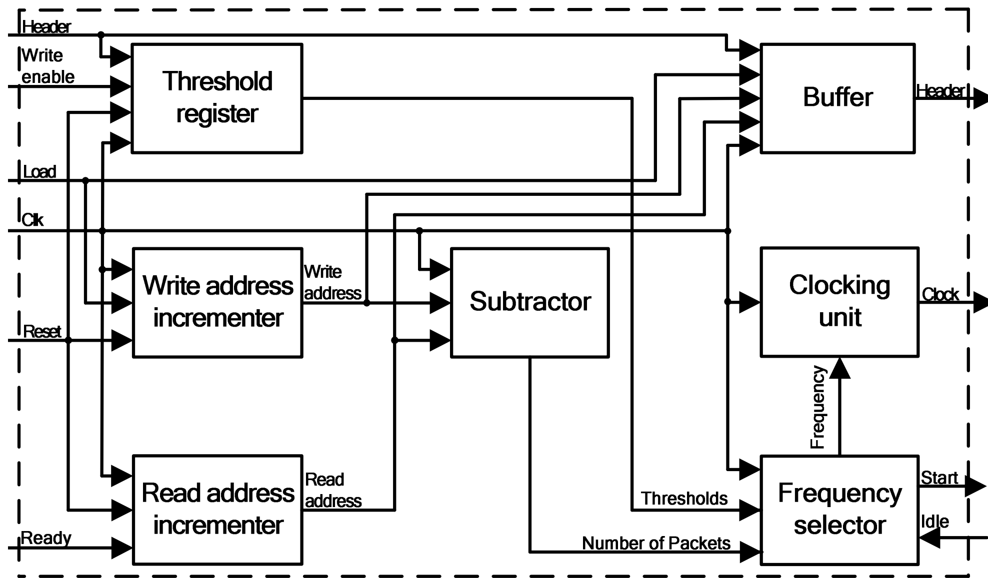


Fig. 4.5. Architecture of the adaptive clocking unit.

output frequency to f_8. The same procedure is followed for state S_8. State S_9 can only be exited if the buffer is cleared, with the lowest-used state S_4 switched to in such a case.

4.4.2 Adaptive Clocking Unit Architecture

The architecture of the ACU is shown in Fig. 4.5. It contains a high speed packet buffer used to capture the fields of a packet's header that are required to classify a packet at a fixed clock speed, ensuring that all packets will be captured. These fields are outputted to the packet classification engine used, with the packets classified on a first come, first served basis. The ACU has an input signal called *Load* that is asserted each time there is a new packet header that requires classification. This *Load* signal has two purposes. It is used as the write enable for the buffer and the enable of a counter that increments the buffer's write address. This write address is the memory location where a packet header is saved. The ACU also contains a second counter that is used to increment the buffer's read address. This counter is incremented each time the classification engine asserts a *Ready* signal, which is used to notify the ACU that it is ready to classify a new packet. The packet classification engine loads a packet header from the memory location specified by the buffer's read address. A subtraction block is used to calculate the number of packets in the buffer. It does this by subtracting the buffer's read address from its write address. The ACU asserts a *Start* signal if this

difference is greater than zero. The *Start* signal is used to notify the packet classification engine that there are packets in the buffer that are ready to be classified. The classification engine will only assert its *Ready* signal and load a new packet header when the ACU's *Start* signal has been asserted.

The ACU also contains a register that stores the threshold values required to determine the clock frequency that the packet classification engine and its memory should be run at. This clock frequency will be the frequency that matches the classifier's processing capacity to the processing needs of the incoming network traffic. These thresholds are inputted into the frequency selector block along with the output of the subtraction block, which indicates the number of packet headers in the buffer. The frequency selector block implements the state machine that was explained in Section 4.4.1, with the aid of Fig. 4.3 and Fig. 4.4. This state machine uses comparators to compare the number of packets in the buffer to the threshold value belonging to each state. The output of this state machine is its current state, which also represents the clock frequency that the packet classification engine and its memory should be run at. This value is outputted to the clocking unit, which contains the PLLs and clock switching logic. The PLLs generate the *N* different clocks that can be used to run the classifier, while the clock switching logic uses the output of the frequency selector block to decide which of these *N* clocks should be used to clock the classifier.

The state machine in the frequency selector block only changes state when the classification engine is in an idle state to prevent problems that could occur due to glitches when the frequency of the classifier's clock is switched. It puts the classification engine in an idle state by placing the *Start* signal low, even if there are packets in the buffer to be classified. This makes the classification engine think that there are no packets to be classified, causing it to enter into its idle state when the packet it may have been processing is classified. The classification engine asserts an *Idle* signal when in its idle state. This is the state where it waits for packets to become available for classification. The state machine in the frequency selector block monitors this *Idle* signal and will only change state when it is asserted. The *Start* signal will be asserted again when the frequency switch has taken place and there are packets in the buffer to be classified. This allows the classifier to continue loading packet headers and classifying packets.

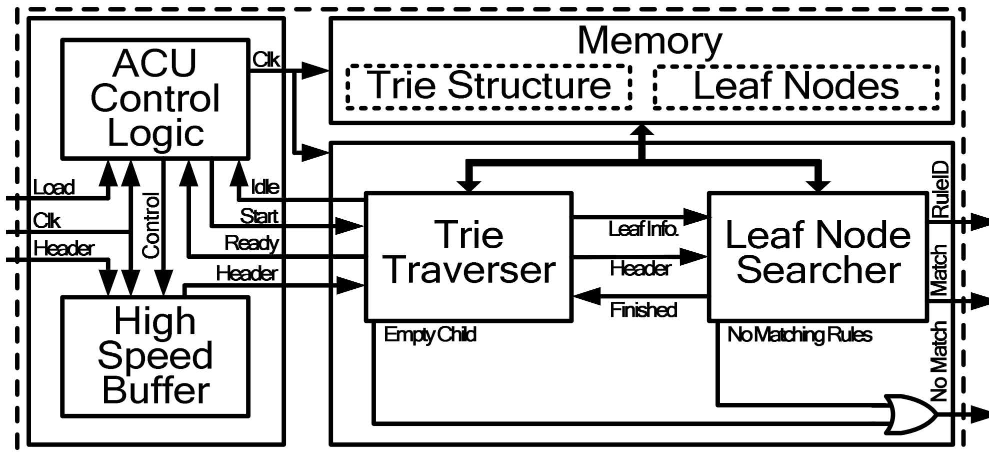


Fig. 4.6. Architecture of low power packet classifier.

4.5 Low Power Architecture for Packet Classification

Fig. 4.6 shows the complete architecture of the low power packet classifier, consisting of the ACU and the packet classification engine that uses on-chip ultra-wide memory. This packet classification engine was explained in detail in Chapter 3. This engine was chosen as it performs best on all types of rulesets when only one engine is available to classify packets, as explained earlier. It can be seen that the architecture of the low power packet classifier is much simpler than the architecture of the classifiers presented in Chapter 3 that were designed to achieve maximum throughput, with a sorter logic block no longer required. The function of a sorter logic block is to make sure that the classification results are outputted in the same order as the order that the packets were buffered when multiple engines are used to classify packets. It is also used to make sure that the matching rule with the highest priority is selected in the case of multiple rule matches between engines.

4.5.1 Hardware Implementation Parameters

The low power architecture for high speed packet classification was implemented in VHDL and targeted at three devices:

- A Cyclone EP3C120F484C8 FPGA, which is built on TSMC 65nm process technology, running at 1.2 Volts.
- A Stratix EP3SE260F1152C47 FPGA, which is also built on TSMC 65nm process technology, running at 0.9 Volts.
- A 65nm ASIC library by TSMC, running at 1.08 Volts.

The low power architecture was synthesised using Altera's Quartus 2 software for both the Cyclone III and Stratix III FPGA implementations. Post place and route timing analysis showed that timing requirements were easily met for the low power packet classifier when it was implemented on both these devices. The ACU met its timing requirement of 128 MHz and the packet classifier met its timing requirement of 32 MHz. The power consumption of these implementations, which is discussed in the next section, was calculated by carrying out post place and route simulations that used the Quartus 2 PowerPlay Power Analyzer Tool to analyse VCD files generated by ModelSim. Section 3.8.3 explained how VCD files are used to measure power consumption in more detail.

For the ASIC solution the logic for the low power packet classifier was synthesised using Synopsys design software. Post place and route timing analysis showed that the timing requirements for both the adaptive clocking logic and packet classification engine logic were again easily met. The Synopsys Prime Power tool was used to analyse the annotated switching information from VCD files generated using ModelSim in order to estimate the power consumption for the logic. The 65nm TSMC RAM compilers were not available to measure the power consumed by the memory used by the ACU and the packet classification engine due to licensing issues. This meant that the power consumption of the memory had to be estimated using 130nm RAM compilers running at 1.2 Volts instead. These RAM compilers were obtained from Chartered Semiconductor Manufacturing.

The power consumption of the memory used by the ACU was estimated using a dual port RAM compiler as it requires read and write memory accesses on the same clock cycle when adding and removing packet headers from the buffer. A single port RAM compiler was used to measure the power consumption of the memory used by the packet classification engine as it can only be accessed by one engine that will perform at most one memory access per clock cycle. The power results for these RAM compilers were normalised so that they were the same as the 65nm process technology running at 1.08 Volts that was used for the low power packet classifier's logic. This was done by using the equation derived in Section 2.6.2 to normalise power consumption when different process technologies and voltages are used.

Table 4.3. FPGA memory and logic utilisation of low power packet classifier.

Device	Logic element usage	Memory usage	f_{max}
Cyclone III	21,641/119,088 (18.2%)	M9Ks 431/432 (99.8%)	32 MHz
Stratix III	14,881/254,400 (5.9%)	M9Ks 859/864, M144Ks 0/48 (52.6%)	32 MHz

Table 4.3 shows the logic and memory usage of the Cyclone III and Stratix III implementations of the low power packet classifier. It can be seen that the logic utilisation is low for both devices as only one packet classification engine is used, with the Cyclone III implementations using 18.2% of its available logic and the Stratix III implementation using 5.9%. The low clock speeds and logic usage made it possible for the designs to be constrained for low power consumption rather than a low area or high clock speeds. This made it possible for even more power savings to be made. The Cyclone III and Stratix III implementations of the low power packet classifier have the same high memory utilisations as the equivalent implementations of the packet classifiers designed for high throughput that were described in Chapter 3. This is because the low power packet classifier is still able to classify packets when using rulesets that contain up to 24,000 rules when using a Cyclone III FPGA and up to 49,000 rules when using a Stratix III FPGA. The ASIC implementation of the low power packet classifier has also been implemented with enough memory to allow it to classify packets using rulesets that contain up to 49,000 rules.

4.5.2 Power Consumption

The power saved by using the ACU in the low power packet classifier was measured by implementing two different systems:

- System A was the low power packet classifier shown in Fig. 4.6. It uses the ACU to buffer incoming packets at a clock speed of 128 MHz while clocking the packet classification engine and its memory at speeds that match the classifier's processing capacity to the processing needs of the network traffic.
- System B used the packet buffer explained in Section 3.7.1 to buffer incoming packets at a clock speed of 128 MHz while clocking the same classification engine and memory used by system A at a fixed clock speed of 32 MHz.

The power consumption of these two systems could then be compared, with the difference being the power saving. Power simulations were run for both systems

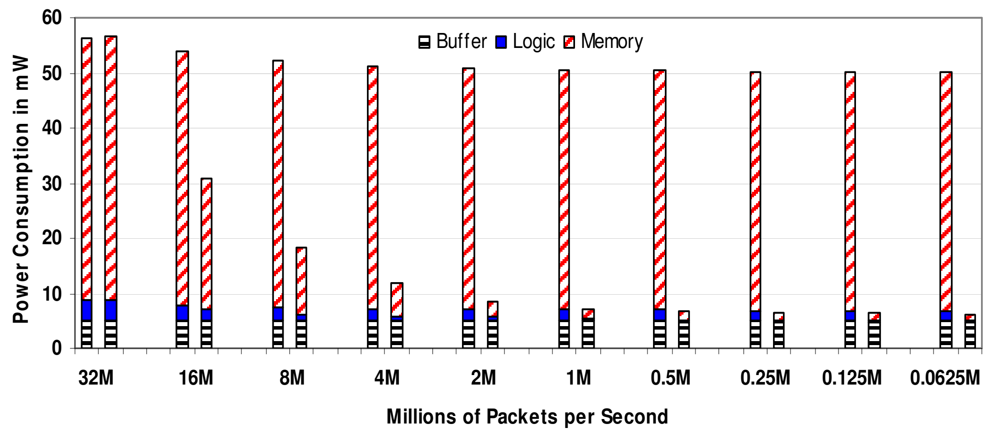


Fig. 4.7. Power used by the ASIC implementation of the low power classifier.

implemented on the Cyclone III and Stratix III FPGAs using the PowerPlay Power Analyzer tool. Power simulations were also run for both systems implemented as an ASIC using the Prime Power tool.

The simulation conditions used when measuring the power consumed by both systems were identical, with packets read in at fixed rates of 32, 16, 8, 4, 2, 1, 0.5, 0.25, 0.125 and 0.0625 Mpps. The exact same packet headers were classified by both systems. Identical search structures were also used by both systems when classifying these packets. The search structure used was built from the synthetic ACL ruleset with 10,000 rules that was created using ClassBench. It requires one memory access at worst to classify a packet. This meant that it was possible for the classifiers in both systems to classify a packet on each clock cycle when reading in 32 Mpps. The power consumption for the two systems implemented on the three technologies can be seen in Fig. 4.7, Fig. 4.8 and Fig. 4.9. The power figures for the low power packet classifier are shown on the right for each packet speed, while the power figures for the classifier that uses a fixed clock speed are shown on the left.

It can be seen by looking at Fig. 4.7 that the low power packet classifier uses 0.25% more power than the classifier that uses a fixed clock speed when it is implemented as an ASIC and used to classify packets at a fixed rate of 32 Mpps. The extra power used is due to the additional logic required by the ACU to enable frequency scaling. The maximum power consumption of the low power packet classifier is 56.48 mW when it is used to classify packets at this speed. At this speed the majority of the power is consumed by the memory used to save the

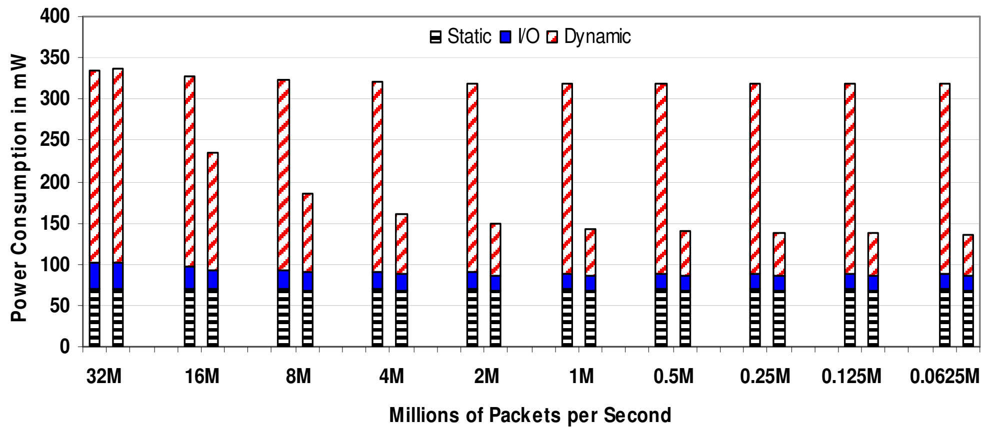


Fig. 4.8. Power used by the Cyclone III implementation of the low power classifier.

search structure. It consumes 84.4% of the total power, with the memory used by the high speed packet buffer consuming 8.8% of the power and the remaining 6.8% used by the logic. Fig. 4.7 also shows that the low power packet classifier uses 89% less power than the classifier that uses a fixed clock speed when the packet arrival rate drops to 0.0625 Mpps. At this rate its power consumption is only 6.24 mW, with most of the power now consumed by the memory used by the high speed packet buffer. It now consumes most of the power as its clock speed is fixed at 128 MHz, while the logic and memory used to save the search structure are run at 0.0625 MHz. The memory used by the buffer consumes 77.9% of the power, followed by the memory used to save the search structure which uses 16% and then the logic which uses 6.1%. The ASIC implementation shows such good power savings as most of the power consumed by it is dynamic rather than static. It can be seen that the power savings flatten out as the packet speeds reach 1 Mpps. This is because the power used by the packet buffer remains steady, leaving little room for a further reduction in power consumption.

Fig. 4.8 shows the power consumption figures for the Cyclone III implementation of the low power packet classifier. It uses 0.7% more power than the classifier that uses a fixed clock speed when packets arrive at a constant rate of 32 Mpps. The extra power used in this implementation is again due to the additional logic required by the ACU to enable frequency scaling. The low power packet classifier's maximum power consumption rises to 333.9 mW when it is implemented on a Cyclone III. At this speed 69.6% of the power consumption is caused by dynamic power, with 20.7% of the power caused by static power and

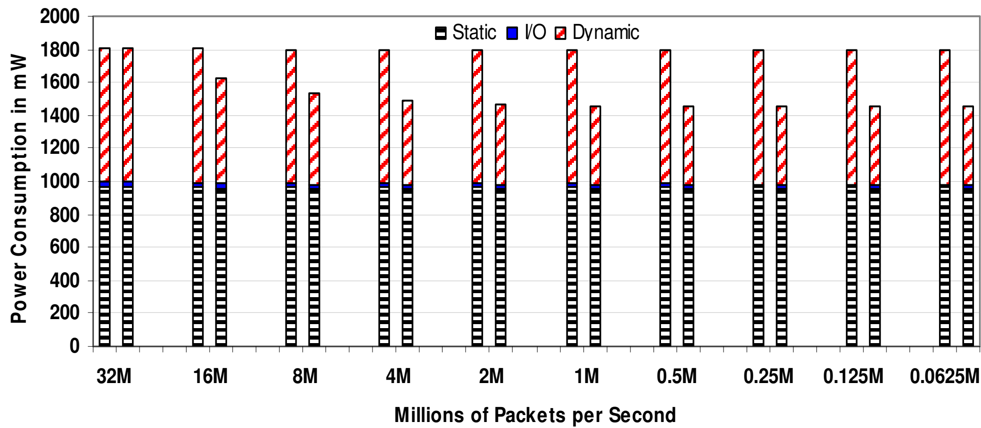


Fig. 4.9. Power used by the Stratix III implementation of the low power classifier.

the remaining 9.7% due to input/output power. The low power packet classifier shows power savings of 57.16% when the packet arrival rate drops to 0.0625 Mpps. At this speed the low power packet classifier consumes 136.36 mW, with 50% of this power now caused by static power, 37.4% caused by dynamic power and the remaining 12.6% caused by input/output power. The power savings for the Cyclone III implementation also flatten as packet arrival rates reach 1 Mpps. This is due to the fact that the static power becomes the dominant cause of power consumption, with frequency scaling only capable of decreasing dynamic power consumption. The Cyclone III implementation shows lower power savings than the ASIC implementation due to the fact that the FPGA has a larger percentage of its power consumption caused by static power.

Finally, Fig. 4.9 shows the power consumption figures for the Stratix III implementation of the low power packet classifier. It can be seen that the power consumed by the adaptive and fixed clock packet classifiers are almost identical when the packet arrival rate is 32 Mpps. This is because the low power packet classifier only requires an extra 0.1% of the Stratix III logic resources to implement frequency scaling. The maximum power consumption of the Stratix III implementation of the low power packet classifier is 1,807 mW when classifying 32 Mpps, with static power causing most of this. Static power makes up 53.3% of the total power consumption, with 44.5% caused by dynamic power and the remaining 2.2% cased by input/output power. The large amount of static power used by Stratix III means that there is reduced scope for power to be lowered through the use of frequency scaling. It can be seen that frequency scaling

achieves a maximum power reduction of 19% as packet arrival rates drop to 0.0625 Mpps. At this speed the Stratix III consumes 1,449 mW, with 66% of this caused by static power, 32.8% caused by dynamic power and the remaining 1.2% caused by the input/output power. This time the power savings flatten as packet arrival rates reach 4 Mpps due to the large amount of static power, which cannot be reduced by frequency scaling. It can be seen from looking at Fig. 4.8 and Fig. 4.9 that the power consumption is much higher for the Stratix III FPGA than the Cyclone III FPGA. This is because the packet classifier implemented on the Stratix III uses double the memory of the classifier implemented on the Cyclone III. The Stratix III also has much more logic and memory resources available, leading to a larger amount of static power consumption.

The power consumption figures presented in this section show that the low power packet classifier is extremely energy efficient even if frequency scaling is not used. The Cyclone III implementation of the low power packet classifier has a maximum power consumption of 333.9 mW when used to classify 32 Mpps. This compares favourably to the similarly sized Cypress Ayama 10128 TCAM-based search engine, which consumes 1,380 mW when used to classify packets at the same rate. It also compares favourably to the Cyclone III implementation of the packet classifier presented in Chapter 3 that was designed to achieve maximum throughput by using two packet classification engines working in parallel. It uses the exact same amount of memory as the low power packet classifier and consumes 488.86 mW when used to classify packets at the same rate.

The ASIC and Stratix III implementations of the low power packet classifier use the same amount of memory and have a maximum power consumption of 56.48 mW and 1,807 mW respectively when used to classify 32 Mpps. This is a large power reduction when compared to the Cypress Ayama 10256 TCAM-based search engine, which has a similar amount of memory and an average power consumption of 2,890 mW when used to classify packets at the same rate. The ASIC and Stratix III implementations of the low power packet classifier also show large power savings when compared to the Stratix III implementation of the packet classifier presented in Chapter 3, which has the same amount of memory. It uses four packet classification engines working in parallel to achieve maximum

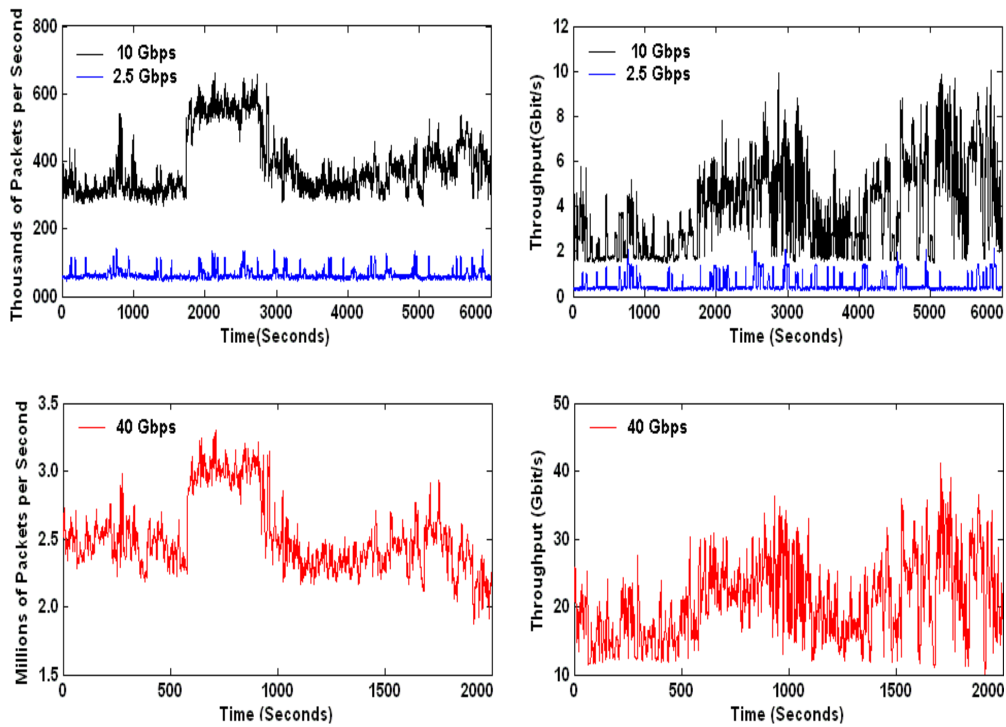


Fig. 4.10. Throughput of the synthetic 2.5 Gbps, 10 Gbps and 40 Gbps packet traces.

throughput. These four engines cause its power consumption to increase to 2,480 mW when it is used to classify 32 Mpps.

4.6 Performance Testing Using Synthetic Traces

The results in the previous section showed the low power packet classifier's power consumption when it is used to classify packets that arrived at fixed rates. It also showed the power savings made at these rates by comparing the low power packet classifier to a classifier that uses an identical classification engine that runs at a fixed clock speed. The results do not, however, show how the low power packet classifier would perform if it was used to classify packets on an edge or a core router operating at 2.5 Gbps, 10 Gbps or 40 Gbps line speeds. This section carries out such an analysis by testing the classifier's performance on synthetic 2.5 Gbps, 10 Gbps and 40 Gbps line speed packet traces, which were created by aggregating Abilene, CENIC, and SCO4 backbone packet traces from the NLANR database until peak line rates of 2.5, 10 and 40 Gbps were reached. These traces can be seen in Fig. 4.10, which shows their throughput both in bits per second and the metric of most interest to the classifier, which is packets per

second. Synthetic traces had to be created to fully test the low power packet classifier because the 2.5 Gbps and 10 Gbps network traces in the NLANR database never got near to their maximum throughput, while 40 Gbps traces are not yet publicly available.

The 2.5 Gbps and 10 Gbps traces created were looked at over a 6,000 second period. The peak throughput in terms of packets per second for these traces is 143,768 p/s for the 2.5 Gbps trace and 661,526 p/s for the 10 Gbps trace. The 40 Gbps trace generated was looked at over a 2,000 second period, with its peak throughput in terms of packets per second being 3,302,488 p/s. This trace was created by compressing the timestamp of the packets aggregated so that they spanned a 2,000 second period rather than a 6,000 second period.

The large number of packets in these traces made it impossible to measure power consumption using the method explained previously, which involves using the packet headers as input stimulus to the low power packet classifier in order to generate VCD files using ModelSim. These VCD files would then be analysed using the Prime Power and PowerPlay power analysis tools. The method which was instead used was to develop a cycle accurate simulator for the low power classifier in C code.

This simulator is similar to the one used in section 4.2.2 to verify that one packet classification engine had enough processing capacity to cope with real network traces. It works by keeping track of the clock frequency that the packet classifier is being run at on any given clock cycle. The simulator estimates the power consumed by the low power classifier on each clock cycle by using the power figures presented in Fig. 4.7, Fig. 4.8 and Fig. 4.9. These figures give the power consumed by the low power packet classifier when it is used to classify packets at different speeds. They were obtained using the Prime Power and PowerPlay power analysis tools, which were used to analyse VCD files generated using ModelSim. The time stamps from the headers of the packets in the 2.5 Gbps, 10 Gbps and 40 Gbps network traces were spliced to the headers used by the ACL, FW and IPC rulesets generated using ClassBench. These traces were then used as input stimulus to the simulator, which classified the packets using the search structures built for the ACL, FW and IPC rulesets.

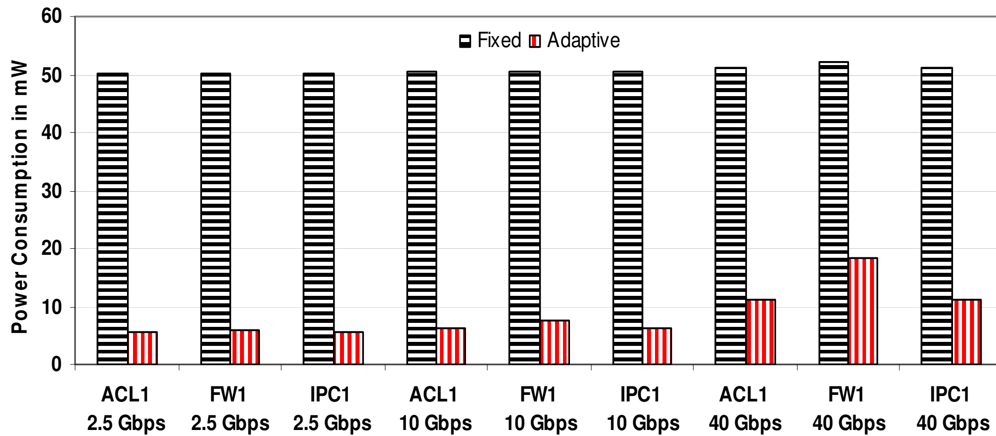


Fig. 4.11. ASIC power usage when classifying packets from synthetic traces.

4.6.1 Power Savings

Fig. 4.11, Fig. 4.12 and Fig. 4.13 show the average power consumed by the ASIC, Cyclone III and Stratix III implementations of the low power packet classifier when they are used to classify packets from the 2.5 Gbps, 10 Gbps and 40 Gbps traces using search structures built for the ACL, FW and IPC rulesets containing 20,000 rules. Appendix A contains graphs that show the power consumed when using the rulesets with 5,000 and 25,000 rules. The results for the rulesets with 20,000 rules are explained in this section because they are the largest rulesets used for testing in this thesis whose search structures are small enough to fit in the on-chip memory of all three devices. These rulesets are also difficult to classify packets for because of their large size.

The power figures for the low power packet classifier are again shown on the right for each trace and ruleset, while the power figures on the left show the power consumed by the classifier that operates at a fixed clock speed in order to show the power saved. Fig. 4.11 shows that the ASIC implementation of the low power packet classifier shows excellent power savings at all line speeds. It reduces power consumption by an average of 88.7%, 86.7% and 73.7% when used to classify packets at 2.5 Gbps, 10 Gbps and 40 Gbps line speeds respectively. It shows such high power savings due to the fact that it does not usually operate at more than a few MHz. This is because of the low throughput of the traces in terms of packets per second due to large average packet sizes and the low number of clock cycles needed to classify a packet. The low power packet classifier shows

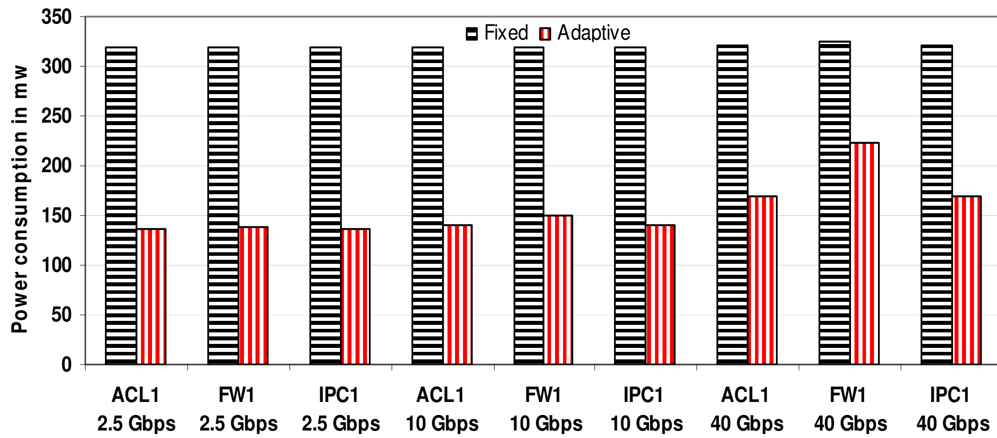


Fig. 4.12. Cyclone III power usage when classifying packets from synthetic traces.

its poorest power saving of 64.6% when used to classify packets at 40 Gbps line speeds for the FW ruleset. This is because it is the line speed with the highest throughput and the ruleset that requires the largest worst case number of memory accesses to classify a packet. The combination of these two factors requires the packet classifier to operate at a higher clock speed, reducing power savings.

It can be seen from looking at Fig. 4.12 that the Cyclone III implementation of the low power packet classifier also performs well across all line speeds, with average power savings of 56.9%, 54.9% and 41.7% when used to classify packets at 2.5 Gbps, 10 Gbps and 40 Gbps line speeds respectively. The Cyclone III implementation shows poorer power savings than the ASIC implementation for two reasons. The first is that the Cyclone III implementation has less memory available, resulting in more clock cycles being needed to classify a packet. The second reason is that a large portion of the power consumed by the Cyclone III is static power, which cannot be reduced by frequency scaling. The poorest power savings by the Cyclone III implementation are again seen when classifying packets at 40 Gbps line speeds for the FW ruleset due to the same reasons explained for the ASIC implementation, with the average power consumption reduced by 31.2%.

Finally Fig. 4.13 shows the power saved when the Stratix III implementation of the low power packet classifier is used to classify packets from real traces. It can be seen that the power savings are much lower than those of the ASIC and Cyclone III implementations due to the fact that the majority of the power consumed is static power. The Stratix III implementation of the low power packet

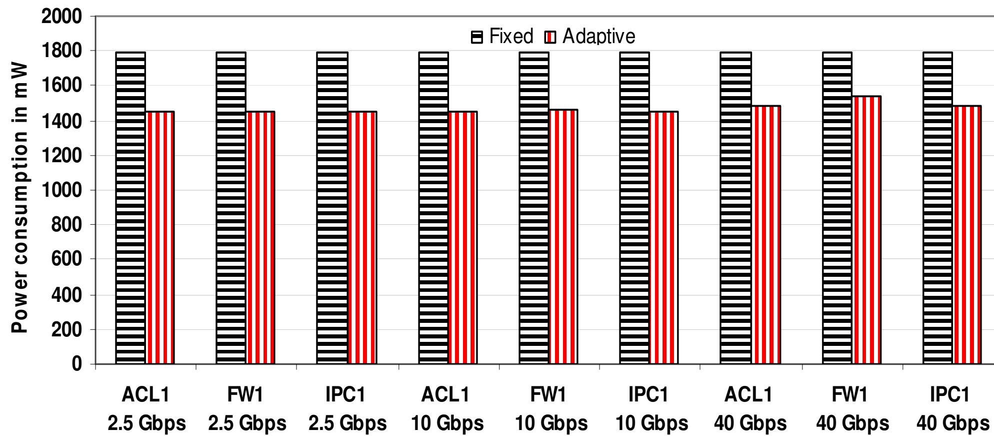


Fig. 4.13. Stratix III power usage when classifying packets from synthetic traces.

classifier reduces power consumption by 19%, 18.6% and 16.1% on average when it is used to classify packets at 2.5 Gbps, 10 Gbps and 40 Gbps line speeds respectively. These power savings are still significant when you consider the tight power budget on a router's line card. The lowest power saving of 14.2% is again made when classifying packets from the 40 Gbps trace using the FW ruleset.

4.7 Summary of Contributions

This chapter has presented a low power packet classifier that is capable of classifying packets at 40 Gbps line rates when using rulesets containing thousands of rules. Its architecture consists of an ACU that dynamically changes the clock speed of an energy efficient packet classifier so that its processing capacity matches the fluctuating processing needs of the network traffic on a router's line card. It does this with the help of a scheme developed to keep clock frequencies at the lowest speed capable of servicing the line card, while keeping frequency switches to a minimum. The low power packet classifier's small logic footprint and low power consumption make it ideally suited to being implemented as an on-chip hardware accelerator relieving the burden from a programmable network processor's processing engines, or as an off-chip high speed classifier on a router's line card.

The ASIC and Stratix III implementations of the low power packet classifier are capable of classifying packets for rulesets containing up to 49,000 rules while its Cyclone III implementation can classify packets for rulesets containing up to 24,000 rules. It has been tested classifying packets from 2.5 Gbps, 10 Gbps and 40

Gbps traces created from real network traces obtained from NLANR while using synthetic rulesets containing up to 25,000 rules. Simulation results show that the low power packet classifier can achieve power savings of between 14-88% if the ACU is used to clock the packet classifier rather than a fixed clock speed.

Chapter 5 - String Matching Architecture

5.1 Introduction

The availability of a hardware accelerator on a router's line card dedicated to the searching of strings/signatures in a packet's payload is essential if networking applications employing DPI are to be moved to the edge or even the core of a network. These applications include network intrusion detection/prevention systems such as Snort [23], which can be used to protect networking equipment and end hosts from the spread and effect of viruses, worms, Denial of Service attacks and other harmful activities. Such attacks can spread rapidly throughout a network, affecting thousands of vulnerable victims in a matter of minutes [21, 22]. Snort can be used to detect and prevent these attacks by searching through the header and payload of the packets passing through an inspection point at wire speed. It searches for known content in packets associated with malicious activity, using a ruleset that contains thousands of rules. The complexity of doing this requires Snort to be implemented in software, limiting its throughput to Mbps [44].

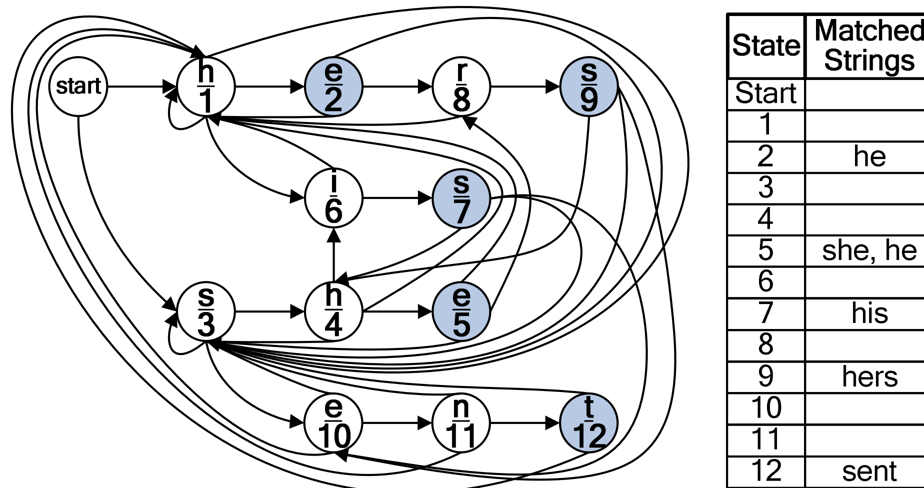
The searching of a packet's payload is the most computationally heavy task in a network intrusion/detection system as the content being searched for could be anywhere in the payload. This means that every byte must be examined to check if any of the thousands of strings being sought are contained within the payload. A new multi-pattern matching algorithm and hardware accelerator are presented in this chapter that can search for the fixed strings contained within rulesets at a guaranteed rate of one character per cycle, independent of the number of strings or their length. This makes it impossible for attackers to flood a system by creating packet payloads on which it performs poorly. The algorithm is an improvement on the Aho-Corasick [37] string matching algorithm. It builds a state machine from the strings being sought, with the state machine used to search the packet payload.

A problem with solutions that use state machines is the large amount of memory required to store the transition pointers used when traversing between states. The algorithm presented here reduces the number of transition pointers that need to be stored at a state by storing a small number of default transition pointers to the states that are most commonly pointed to in a lookup table. These default transition pointers can be shared by all states, dramatically reducing memory usage.

The rest of this chapter is organised as follows. Section 5.2 explains the operation of the multi-pattern matching algorithm Aho-Corasick. This is explained so that the improvements presented can be better understood. Section 5.3 describes why the characteristics of rulesets used for network intrusion detection/prevention systems allow for large memory savings when default transition pointers are used. It also explains how default transition pointers can be used and the steps involved in building the search structure. The memory organisation of this search structure and the architecture of the hardware accelerator designed to use it are presented in Section 5.4. Performance results are presented in Section 5.5, showing the memory savings made from using default transition pointers on different sized rulesets, the throughput of the hardware accelerator using the search structures built from these rulesets and the hardware accelerator's power consumption. The characteristics of the strings from the Snort rulesets used to test the algorithm and hardware accelerator are also presented in this section. It also compares their performance with the performance of other state of the art hardware-based approaches used to implement string matching. Section 5.6 concludes this chapter.

5.2 String Matching Using Deterministic Finite Automaton

The Aho-Corasick algorithm matches multiple strings using a deterministic finite state machine, which is also known as Deterministic Finite Automaton (DFA). The DFA has a start state from which all strings to be matched are extended. The start state is the state where no strings have been partially matched. The strings to be matched extend from the start state one state per character. Strings are added sequentially to the state machine, with strings that share a common stem also sharing a number of common states extending from the start state. To match a string against a payload the search begins at the start state and traverses from one state to another based on transitions decided by the input characters. Each state in



State	Matched Strings
Start	
1	
2	he
3	
4	
5	she, he
6	
7	his
8	
9	hers
10	
11	
12	sent

Fig. 5.1. Aho-Corasick state machine showing transition pointers and matched states.

the state machine will store its transition pointers and the number of the strings that will have been matched if the state is entered. A state's depth in the state machine is the fewest number of transitions needed to reach it from the start state.

The Aho-Corasick algorithm proposes two methods for storing transition pointers, with one solution using a failure function and the other a move function. Each solution will have the same worst case number of transition pointers, which may need to be stored at a state. This is equal to the number of characters in the ASCII code, of which there are 256. The solution that uses the failure function requires the lowest amount of memory on average but cannot guarantee the processing of one input character on each clock cycle. This is because each state only stores the transitions for characters whose next state has a depth one level higher than the depth of the current state. All other characters must follow a fail transition, which will cause a wasted transition. Multiple fail transitions may have to be followed until the correct state is found, wasting many cycles.

The second approach, on which the new algorithm is based, uses a move function. In this approach each state stores the transitions for all states that could be transitioned to regardless of their depth in the state machine. This means that there is no need for a fail function and thus no wasted transitions, so that a new input character can be processed on each clock cycle. The disadvantage of this approach is that it uses larger amounts of memory to store all possible transition pointers.

Fig. 5.1 shows a state machine constructed from the strings (he, she, his, hers, sent). The state machine does not use failure pointers, storing all possible transition

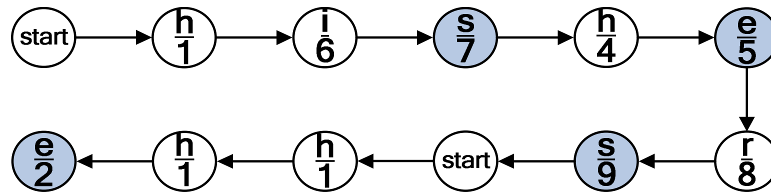


Fig. 5.2. Sequence of strings that will be traversed if text (hishersqhhe) is searched.

pointers instead. This allows each byte from a packet's payload to be processed in a single clock cycle. Each state is represented by a circle, with the two values inside each circle indicating the input character required to transition to that state (the new algorithm presented here calls this the state's character value) and the state's number. All valid transition pointers are shown for each state, apart from the transition pointers that point to the start state. All states in this state machine have a transition pointer that points to the start state. This transition pointer is followed when a character is inputted from the payload being searched that has no full or partial string match. A shaded state indicates a state where a string or strings will have been matched if it is entered. A table is shown in Fig. 5.1 that lists all states where strings will have been matched if they are entered and the corresponding matched strings.

The sequence of states that will be traversed if the text (hishersqhhe) is searched can be seen in Fig. 5.2. It shows that it takes one clock cycle to traverse each input character. This is true for all possible input character sequences that could be searched. A guaranteed throughput makes this type of state machine ideally suited to carrying out DPI for network intrusion/detection systems as it can guarantee a specific line rate. This is important as it ensures that no packets will be able to make their way through the network without being inspected.

5.3 Memory Reduction

The storage of transition pointers is the largest cause of memory usage when saving a state machine used for DPI. This is because each state has to store the 256 pointers needed to represent all possible character transitions unless some kind of memory compression scheme is used. Even only storing the pointers that point to a state other than the start state can lead to large memory usage. This section explains the scheme developed which reduces the amount of transition pointers that need to be stored at a state.

5.3.1 DFA Memory Usage Observations

The transition pointers of the states at all depths in a state machine used to perform DPI for intrusion detection prevention/systems such as Snort mainly point to a few states near the start of the state machine. This is because of the way in which the state machine is constructed, with the strings being sought extending from the start state one state per character, meaning that the majority of states will only have one forward pointing transition pointer. The majority of the transition pointers will point to states with a depth less than the current state. For example, a series of input characters could mean traversing to a state with a depth of twelve in the state machine. There will typically only be one character at this state that would mean traversing deeper into the state machine, with all remaining characters resulting in a traversal backwards to a partial match of another string. The depth of the state transitioned to will be equal to the length of the partial match.

There is a wide variation among the strings contained within the rules used by the rulesets of intrusion detection/prevention systems such as Snort. This means that partial matches are usually small, so transition pointers pointing backwards in the state machine will normally point to a state with a low depth. A state machine built for the Snort rulesets with 6,275 strings, using the Aho-Corasick algorithm, will contain 109,467 states. These states will store a total of 9,524,131 transition pointers that point to states other than the start state, with 78% of these transition pointers pointing to states with a depth of one in the state machine, 15% pointing to states with a depth of two, 4% pointing to states with a depth of three and the remaining 3% pointing to states with a depth greater than three.

A state machine built to search for thousands of strings will only have a few hundred states in the heavily pointed to area near the start of the state machine. This is due to the congested nature of the area near the start, where many strings share common states. A large reduction in memory usage can be achieved by removing the transition pointers that point to the same few states near the start and placing them in a small lookup table where they can be shared by all states. These transition pointers placed in the lookup table are called default transition pointers. The number of transition pointers that need to be stored in the states of the state machine is reduced by over 98% in the Snort ruleset used for testing. This is

achieved by placing default transition pointers to the most commonly pointed to states at a depth of one, two and three in a lookup table.

5.3.2 Insertion of Default Transition Pointers

Default transition pointers are used to reduce the amount of memory needed to save a state machine without affecting the number of transitions needed to traverse it when searching through a packet's payload. To do this a lookup table containing 256 memory words is used. Each memory word stores the default transition pointers for one of the 256 ASCII characters. Each ASCII character has default transition pointers to states with its character value at depths of one, two and three in the state machine.

Default Transition Pointers to States at a Depth of One

The maximum number of states that can occur at a given depth in the state machine is 256^d, where d is the depth. This means that it is possible to store a default transition pointer to all states at a depth of one in the lookup table as there can only be a maximum of 256. ASCII characters that have a state with its value at a depth of one in the state machine will store the address of this state as its default transition pointer, while ASCII characters who do not have a state with its value at a depth of one in the state machine will store the address of the start state as its default transition pointer.

A state will only store transition pointers to states that do not contain a default transition pointer in the lookup table. Each transition pointer stored in a state will require two pieces of information. The first piece of information is the character value needed to follow the transition pointer, and the second piece of information is the memory address of the state being pointed to. An input character will need to perform the following steps when traversing from one state to another. It begins by retrieving the information belonging to its default transition pointer stored in the lookup table. The information belonging to the current state is then analysed. A transition pointer stored at the current state is followed if one exists for the current input character, otherwise the default transition pointer retrieved from the lookup table is followed.

Fig. 5.3 (A) shows how the state machine in Fig. 5.1 looks after the insertion of default transition pointers to states at a depth of one. It also shows the resulting

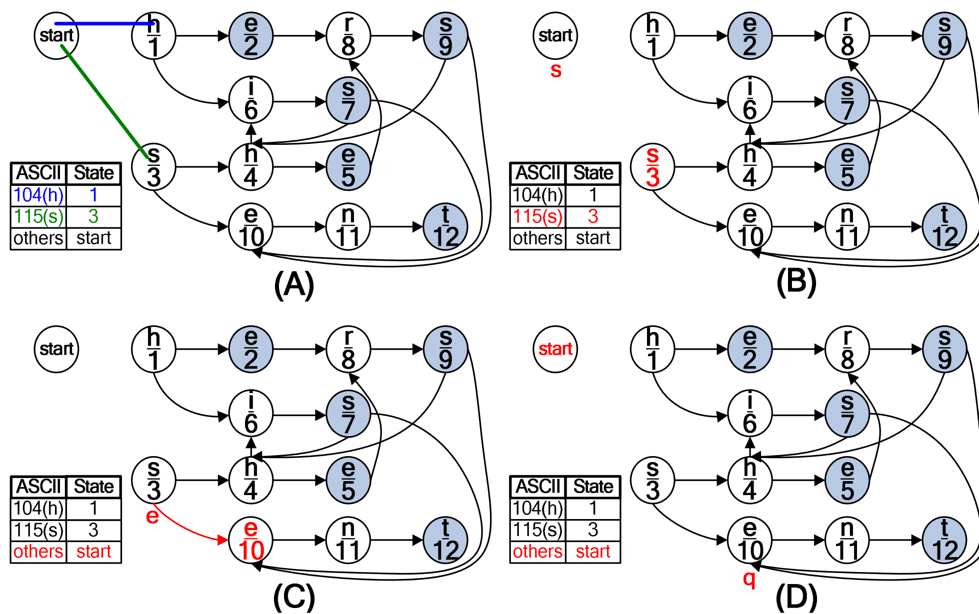
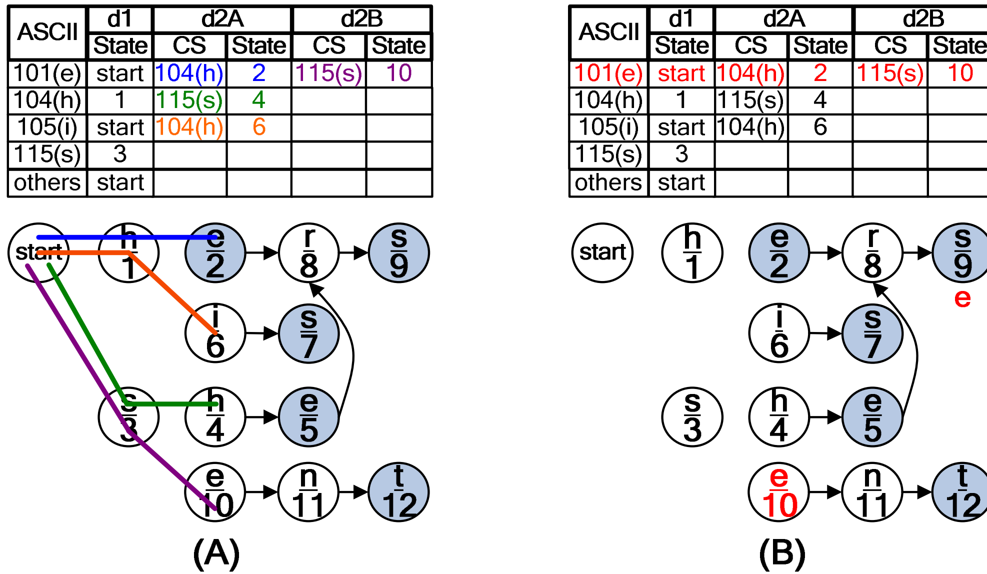


Fig. 5.3. Use of default transition pointers to states at a depth of one.

lookup table. It can be seen that even only using default transition pointers to states at a depth of one can have a large effect on memory usage, reducing the average number of transition pointers stored at a state from 2.846 to 1.231, which is a reduction of 57%. Fig. 5.3 (B), Fig. 5.3 (C) and Fig. 5.3 (D) show how the state machine and default transition pointers to states at a depth of one are used to search the text (seq).

Fig. 5.3 (B) shows that the first input character (s) will start at the start state (the state where there are no partially matched strings). It can be seen that the start state stores no transition pointers as it can only point to states at a depth of one in the state machine, with all of these states having default transition pointers. The input character (s) will use its default transition pointer returned from the lookup table to transition to state 3. Fig. 5.3 (C) shows the transition made by the next input character (e). It can be seen that state 3 stores a valid transition pointer for the input character (e), which means that the default transition pointer character (e) retrieved from the lookup table does not need to be followed. The valid transition pointer points to state 10. Fig. 5.3 (D) shows the transition made by the final input character (q). There is no valid transition pointer stored at state 10 for (q), which means that it must use its default transition pointer returned from the lookup table. This default transition pointer points to the start state as there are no partially matched strings.



ASCII	d1	d2A		d2B	
	State	CS	State	CS	State
101(e)	start	104(h)	2	115(s)	10
104(h)	1	115(s)	4		
105(i)	start	104(h)	6		
115(s)	3				
others	start				

ASCII	d1	d2A		d2B	
	State	CS	State	CS	State
101(e)	start	104(h)	2	115(s)	10
104(h)	1	115(s)	4		
105(i)	start	104(h)	6		
115(s)	3				
others	start				

Fig. 5.4. Use of default transition pointers to states at a depth of two.

Default Transition Pointers to States at a Depth of Two

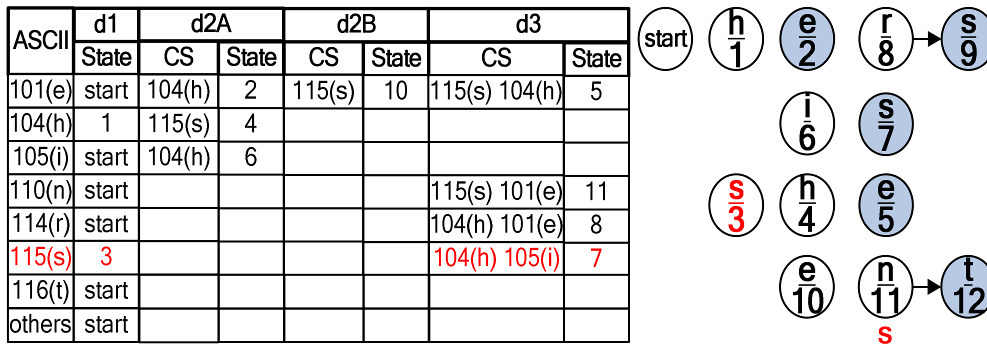
A large percentage of states will also store transition pointers to states at a depth of two in the state machine because they are close to the start. Storing a default transition pointer to all possible states at this depth would not be memory efficient as 65,536 of them would need to be stored. The lookup table therefore only stores default transition pointers to the four most commonly pointed to states for each character value at this depth. It was found through testing of strings used in the Snort ruleset that four was the optimum value as it resulted in the smallest amount of memory being needed to store the state machine and lookup table. Default transition pointers pointing to states at a depth of two require two pieces of information. The first piece of information required is the memory location of the state pointed to and the second piece of information is the character value of the state that connects this state to the start state. The character value of the state that connects it to the start state is needed because there can be multiple states at a depth of two with the same character value. The character value of the state that connects it to the start state is used to distinguish which state at a depth of two is being pointed to.

Fig. 5.4 (A) shows how the state machine in Fig. 5.1 looks after the insertion of default transition pointers to states at a depth of two and one. It also shows how the default transition pointers to the four states at a depth of two and two states at

a depth of one are stored in the lookup table. The columns labelled CS (Connecting State) show the character value of the state that connects the start state to the state pointed to at a depth of two. It can be seen in this example that the ASCII character (e) has two default transition pointers to states at a depth of two. This is because there are two states at this depth with the character value (e). These states are states 2 and 10. The default transition pointers to these states are distinguished by the character value of the state that connects them to the start state. Here a state with the character value (h) connects state 2 to the start state, while a state with the character value (s) connects state 10 to the start state.

An input character will now need to perform the following steps when traversing from one state to another if transition pointers to states at a depth of one and two are used. The first step involves the input character retrieving its default transition pointers from the lookup table. It can retrieve a maximum of five default transition pointers, with one of these pointing to the start state or a state at a depth of one and the remaining four pointing to states at a depth of two. These default transition pointers will be analysed if no valid transition pointer at the current state is found. The default transition pointers to states at a depth of two are analysed first. This is done by comparing their CS values to the character value of the current state (value of the previous input character). A default transition pointer to a state at a depth of two is followed if there is a match, otherwise the default transition pointer pointing to the start state or a state at a depth of one is followed.

Fig. 5.4 (B) shows an example of how the default transition pointers to states at a depth of two are used. In this example the previous input character (s) has transitioned to state 9. The new input character (e) begins by retrieving its default transition pointers from the lookup table. It then checks the current state for a valid transition pointer that it can follow. There is none in this case so it analyses the default transition pointers to states at a depth of two. Two such pointers exist, with one pointing to state 2 and the other to state 10. The value of the current state is compared to the CS value for each of the default transition pointers to states at a depth of two. The default transition pointer that points to state 10 is followed because its CS value matches.



ASCII	d1	d2A		d2B		d3	
	State	CS	State	CS	State	CS	State
101(e)	start	104(h)	2	115(s)	10	115(s) 104(h)	5
104(h)	1	115(s)	4				
105(i)	start	104(h)	6				
110(n)	start					115(s) 101(e)	11
114(r)	start					104(h) 101(e)	8
115(s)	3					104(h) 105(i)	7
116(t)	start						
others	start						

Fig. 5.5. Use of default transition pointers to states at a depth of three.

Default Transition Pointers to States at a Depth of Three

States at a depth of three in the state machine will be pointed to far less often than the states that precede it. However, through testing it was found that significant memory savings can be made by saving one default transition pointer to the most commonly pointed to state for each character at a depth of three. Default transition pointers to states at a depth of three require three pieces of information. The first piece of information required is the memory location of the state pointed to and the second and third pieces of information are the character values of the two states that connect the start state to the state pointed to. Again these character values are needed to distinguish the state pointed to at a depth of three from other states at this depth that can have the same character value.

An input character will now have to check if it can follow the default transition pointer to a state at a depth of three before it can consider following a default transition pointer to a state at a depth of two or one. These default transition pointers need to be checked in the case where there is no valid transition pointer that can be followed from the current state. Fig. 5.5 shows how the state machine in Fig. 5.1 looks after the insertion of default transition pointers to states at a depth of three, two and one. It also shows the complete lookup table. The use of default transition pointers to states at a depth of three, two and one reduces the average number of transition pointers stored at a state in this example from 2.846 to 0.154, which is a reduction of 95%. The maximum number of transition pointers that need to be stored at a state has also been reduced from four to one. Reductions of this magnitude result in massive savings in memory usage as

rulesets scale to contain thousands of strings, with these strings ranging in length from a few bytes to a few hundred bytes. The large reduction in the number of transition pointers that need to be stored at a state also allows the logic used to traverse the state machine to be simplified as only small amounts of data need to be processed during each state traversal.

Fig. 5.5 also shows an example of how to use the complete lookup table and state machine to traverse from one state to another. In this example the previous input character (n) has transitioned to state 11. The new input character (s) begins the process of traversing a state by retrieving its default transition pointers from the lookup table. It then checks to see if there is a valid transition pointer stored at the current state that it can follow. There is none in this case which means that the default transition pointer to the state at a depth of three must be analysed next. This is done by comparing the previous two input characters to the default transition pointer's CS value (character values of the states that connect the start state to the state pointed to). These values do not match as the previous two input characters were (e) and (n), with the character values needed to follow the transition pointer being (h) and (i). This means that the default transition pointers to states at a depth of two must be analysed next. It can be seen that the character (s) has no default transition pointers to states at this depth. This means that the final default transition pointer that points to state 3 must be followed.

5.3.3 Algorithm for Building Search Structure

This section explains the steps that need to be taken when building the state machine and lookup table required to search a packet's payload for specific strings. There is only one user defined constraint that needs to be specified before the building of the state machine and lookup table can begin. This constraint is the maximum number of transition pointers that may be stored at a state. This constraint is used because the string matching hardware accelerator explained in Section 5.4.2 has been designed to handle a maximum of thirteen transition pointers at each state in order to simplify the logic needed and to reduce the amount of memory required to store a state, as explained in Section 5.4.1. The capacity to store a maximum of thirteen transition pointers at each state is more

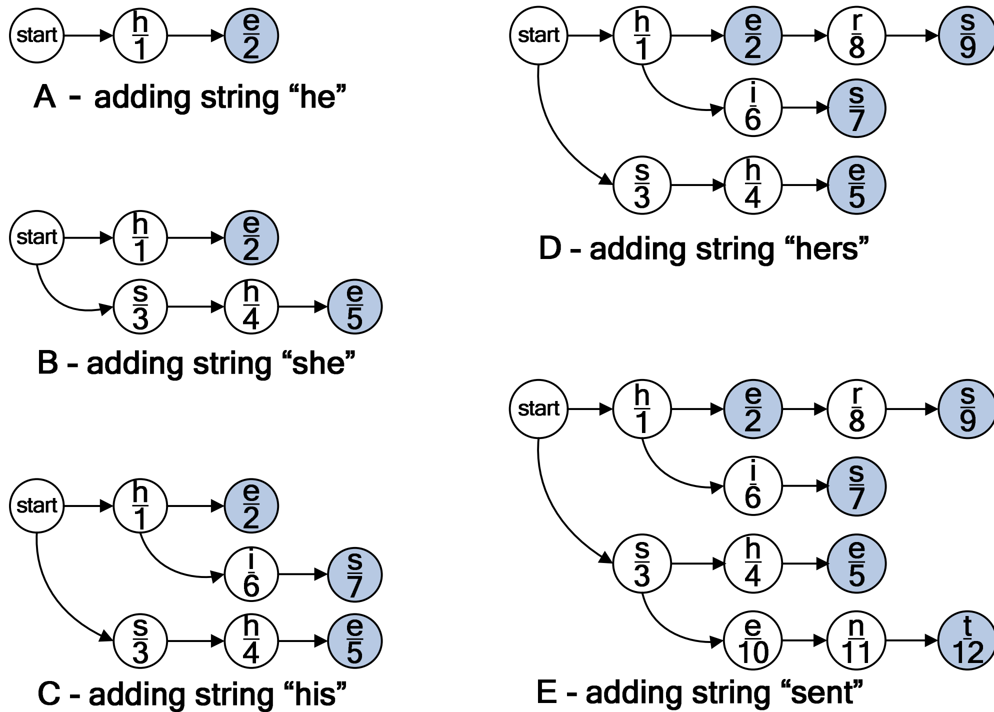


Fig. 5.6. Recording a state's depth, character value and forward pointing transitions.

than enough due to the large memory reductions achieved through the use of default transition pointers.

The first step involves recording the states used in the state machine along with their depth, character value (ASCII value of the input character needed to transition to it) and forward pointing transitions (transition pointers that point to a state whose depth is one greater than the depth of the current state). Fig. 5.6 shows a step by step example of how this is done for the state machine shown in Fig. 5.1. It is done by extending each string to be matched from the start state one character at a time. Each character will have a state, with strings that share common stems also sharing common states. The forward pointing transition pointers are recorded when laying down each string one character at a time. A state's depth is the shortest number of transitions taken to reach it from the start state.

Step two records the remaining transition pointers for each state (transition pointers that point to a state whose depth is equal to or less than the depth of the current state). Fig. 5.7 helps to explain how this is done by showing how the transition pointer for character (h) is recorded in state 9. The transition pointer for

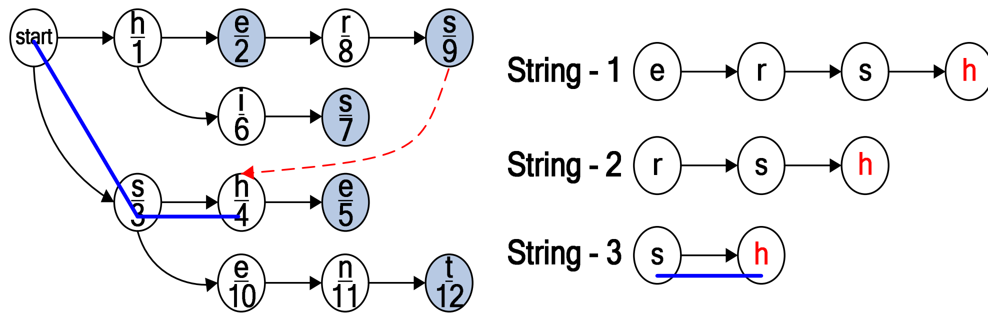


Fig. 5.7. Recording a state's non-forward pointing transitions.

each ASCII character at a state that does not already have a transition pointer is calculated by first forming a string that is made up of:

- The character that the transition pointer is being calculated for (h in Fig. 5.7).
- The character value of the current state (s in Fig. 5.7).
- The character values of the states connecting the current state to the start state minus the character value of the state nearest the start state (e and r in Fig. 5.7).

This string is checked against the character value of the other states (and the character values of the states that connect them to the start state) whose depth is equal to the string length. A match will mean placing a transition pointer to the matched state. No match will mean shortening the string by dropping the first character value and re-matching the string to states whose depth is equal to the length of the new string. This process continues until a state matches or the string can no longer be shortened, which will lead to the start state being pointed to.

In Fig. 5.7 the string *ersh* is compared to the character values of state 12 and the states that connect it to the start state, as the depth of state 12 is equal to the length of the string. These values do not match so the string *ersh* is then compared to the character values of state 9 and the states that connect it to the start state, as the depth of state 9 is also equal to the length of the string. This does not match either so the first character is dropped to form the string *rsh*. This string is then compared to the character values of state 8 and the states that connect it to the start state, as the depth of state 8 is equal to the length of the new string, with no match. The same is also done for states 7, 5 and 11 as they are also at the same depth, with no match. The first character is again dropped, creating the string *sh*. This string matches the character values of state 4 and the state that connects it to

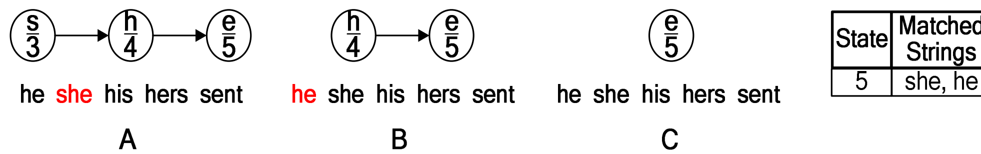


Fig. 5.8. Recording the strings matched if a state is entered.

the start state, which means that the transition pointer for character (h) will point to state 4 as shown by the dashed arrow.

In the third step each state records the number of the strings that will have been matched if it is entered. This is done by making a string for each state comprised of the character values of the state and the states that connect it to the start state. This string and shortened versions of it made by dropping the first character are compared to the list of strings being sought. Matching string numbers will be recorded in the state. Fig. 5.8 shows how state 5 records the strings that will have been matched if it is entered.

The remaining steps explain how default transition pointers are inserted in order to reduce memory usage. Default transition pointers are inserted to states at a depth of one first, then to states at a depth of three and then to states at a depth of two, with the following steps explaining why this is done. Detailed diagrams have been given in Section 5.3.2 that explain how default transition pointers are inserted.

Step four is where the default transition pointers to states at a depth of one are inserted. To do this each of the possible 256 states at this depth have their state number placed in the lookup table. The position of each state number in the lookup table is equal to its ASCII character value. Any position not filled in the lookup table will mean that no state with its ASCII character value exists at a depth of one. This means that a pointer to the start state will need to be placed here. Transition pointers to states at a depth of one are then removed from all states in the state machine.

The fifth step is where the default transition pointers to states at a depth of three are inserted. This is done by first counting how many times each state at a depth of three is pointed to. Default transition pointers to the most commonly pointed to states for each ASCII character will be inserted in the lookup table. Transition pointers to the states chosen at this depth are then removed from all states in the state machine.

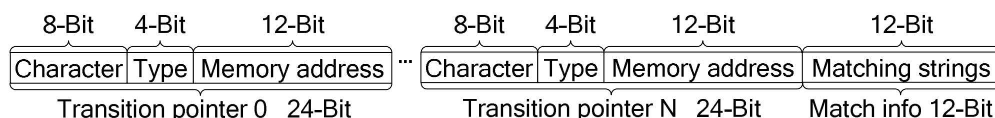


Fig. 5.9. Memory organisation of information needed to store a state.

The sixth step is where the user defined limit on the maximum number of transition pointers that can be stored at a state is used. States that exceed this limit (if any) are first selected. The four most commonly pointed to states (by the states exceeding the threshold) at a depth of two for each ASCII character are first placed in the lookup table. Transition pointers to the states chosen are then removed from all states in the state machine. Four default transition pointers may not have been used for each ASCII character. This will depend on how many states were pointed to by states exceeding the user defined threshold (if any). Space for any unused default transition pointers at a depth of two will be filled by counting the most commonly pointed to states at a depth of two not already in the lookup table and inserting them in the lookup table. This is done until the lookup table is full or there are no more states at a depth of two that require default transition pointers.

5.4 Memory Organisation and Hardware Architecture

5.4.1 Memory Layout

The hardware accelerator has been designed to handle states containing up to 13 transition pointers. Most states, however, will contain less than two transition pointers on average after the insertion of default transition pointers, making it wasteful to allocate the same amount of memory for all states. The hardware accelerator has therefore been designed to handle 15 different state types. A states type indicates how many pointers it has and its position in a memory word. State types 1-9 are used to store states containing 0-1 transition pointers, types 10-12 store states containing 2-4 transition pointers, type 13 stores states containing 5-7 transition pointers, type 14 stores states containing 8-10 transition pointers while type 15 stores states containing 11-13 transition pointers.

Fig. 5.9 shows the number of bits required to store a state's transition pointers and matching string information. Each transition pointer stored at a state will require 24 bits, with 8 bits being used to store the character value needed to follow the

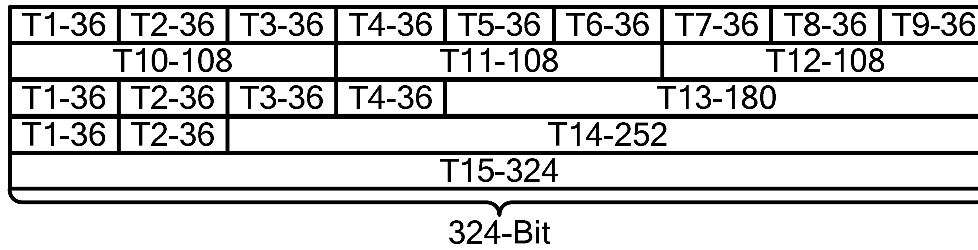


Fig. 5.10. Possible positioning of the state types in memory and their bit size.

pointer. Another 12 bits are used to store the address of the state being transitioned to and 4 bits to indicate its type. The string numbers that may have been matched when a state is entered are stored in a memory block separate to the one used to store the states used by the state machine. This is done to ensure that the fetching of a state's matching string numbers does not reduce throughput when traversing the state machine. Each state uses 12 bits to indicate if any strings have been matched when it is entered and if so the location of these matching string numbers. The block of memory used to store a state's matching string numbers is 27 bits wide. Each memory word holds two 13-bit string numbers and a flag bit. A state will point to the memory word where its matching string numbers are stored. These string numbers are outputted two at a time, with the flag bit used to indicate when all matching string numbers have been outputted.

The number of bits required to store a state ranges from 36 for states containing 0-1 transition pointers to 324 for states containing 11-13 transition pointers. The memory words used to save these states must therefore be 324 bits wide to ensure that the information needed to traverse all states can be accessed in a single clock cycle. The states used by the state machine will be a variety of different sizes, so it is important that they are carefully arranged in memory after the state machine has been built to prevent gaps of unused memory. Fig. 5.10 shows where the different state types can be positioned in a memory word and the amount of space in bits that they occupy. State types 15, 14 and 13 are first saved to memory. These state types are rare due to the memory reduction techniques used. The storage of state types 14 and 13 will leave gaps of unused memory. States containing 0-1 transition pointers are used to fill these gaps as they are the most commonly used state in the state machine. The next step involves storing states that contain 2-4 transition pointers, with each memory word being able to store three such states. The final step stores the remaining states containing 0-1 transition pointers nine at

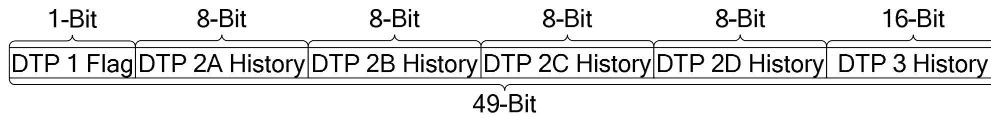


Fig. 5.11. Organisation of a lookup table memory word.

a time to each memory word. This results in the states being saved to memory in the most efficient way, with no gaps of unused memory.

The amount of bits required to save the memory words used by the lookup table can be reduced from 136 to 49 by saving the states pointed to by the default transition pointers for each ASCII character at a fixed memory location and making all states pointed to the same type. Each ASCII character can have a maximum of six default transition pointers to states with its character value spread across depths of one, two and three. These states are always saved in the same six memory locations and saved as type 15 states that can store up to 13 transition pointers. These memory locations can be used to save other states not pointed to by a default transition pointer in the event that any of an ASCII character's six default transition pointers are not used. These default transition pointers might not be used because a specific ASCII character might not have states with its value in the state machine at depths of one, two or three. Not needing to save the address or type of the state pointed to in the lookup table saves 16 bits for each default transition pointer.

The organisation of a lookup table memory word can be seen in Fig. 5.11. The default transition pointer for each ASCII character that points to a state at a depth of one will require one bit. This bit is used to specify if a state exists at a depth of one for this ASCII character. The existence of this state will mean traversing to it, while its non-existence will mean traversing to the start state. The four default transition pointers to states at a depth of two for each ASCII character will require eight bits each to store the character value of the state that connects the state being pointed at to the start state. The default transition pointers to states at a depth of three require 16 bits to store the character values of the two states that connect the state being pointed at to the start state.

5.4.2 Hardware Accelerator Architecture

The hardware accelerator has been designed to use multiple string matching blocks on the same FPGA. The Stratix III implementation uses six string matching

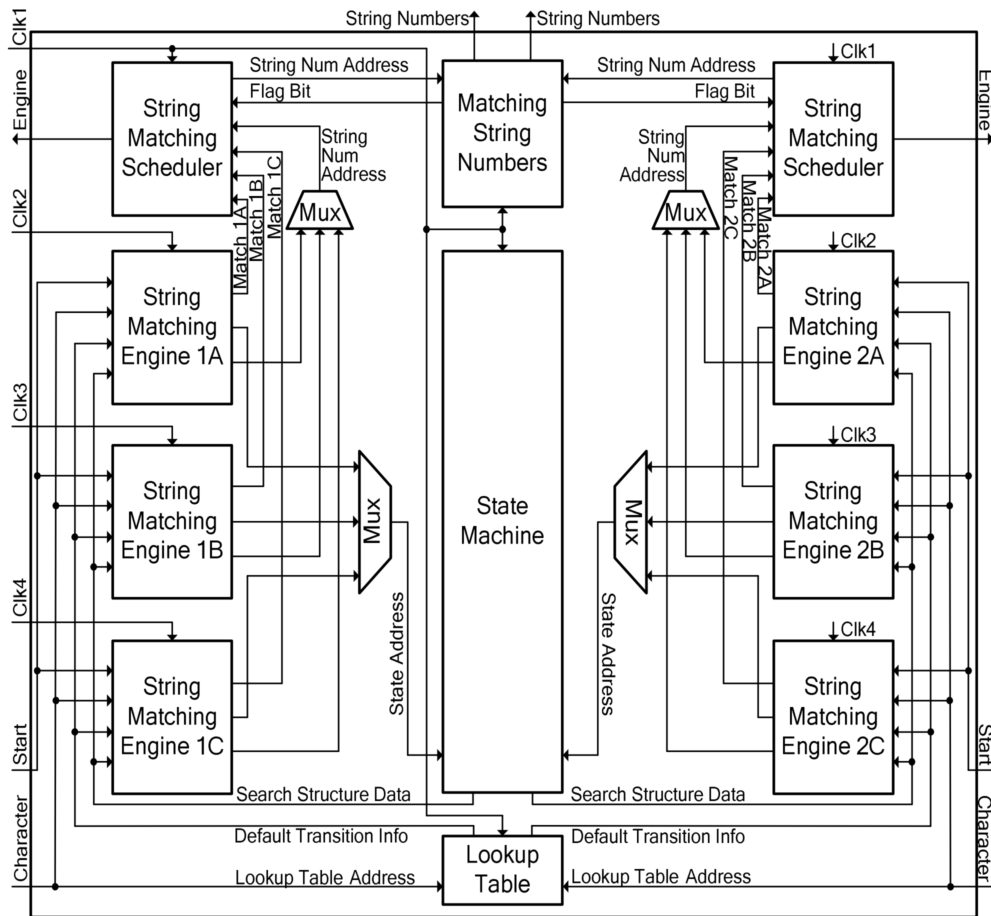


Fig. 5.12. Architecture of a string matching block.

blocks to achieve a throughput of over 40 Gbps, while the implementation on the smaller low power Cyclone III uses four string matching blocks to achieve a throughput of over 10 Gbps. The architecture of a string matching block can be seen in Fig. 5.12. Each string matching block contains six string matching engines, which means that the Stratix III implementation has 36 engines in total and the Cyclone III implementation has a total of 24 engines. Each string matching block has its own memory, which means that the Stratix III implementation can store up to six DFAs and the Cyclone III implementation can store up to four DFAs.

For rulesets containing many thousands of rules the strings being sought can be broken into different groups, with a different DFA built for each group. Each DFA can be stored to a separate string matching block. This gives the string matching blocks the ability to work in parallel on the same packet, with each

string matching block searching for a subset of the strings. A single DFA can be built for smaller rulesets. Saving this DFA to all string matching blocks gives them the ability to work individually, allowing them to search for all strings in a packet so that the highest possible throughput can be achieved.

A string matching block uses true dual port memory to store the matching string numbers, state machine and lookup table in order to maximise throughput. Three engines share access to each of the data ports belonging to the memory used to save the state machine. The string matching engines search the payloads of the incoming packets for matching strings, using information from the lookup table and memory used to save the state machine. A string matching block also has two string matching schedulers, with each scheduler using a data port of the memory used to save the matching string numbers. Each scheduler is used to retrieve the matching string numbers from memory for the three string matching engines sharing a data port.

Three string matching engines share a data port as the maximum clock speed of each engine is slower than the maximum clock speed that memory can obtain. This is due to logic delays in the string matching engines. The memory runs at a speed equal to three times that of an engine. Each engine sharing a port runs at the same clock speed, with the clock for each engine 120º out of phase with the clock of the previous engine. This allows for a simple memory interface as the read commands for the three engines can simply be multiplexed together, with each engine having access to 33% of the memory's bandwidth. Each engine is used to process a separate packet, meaning that six packets are needed to keep the memory in a string matching block fully utilised.

The bytes for the packets being searched by the three engines sharing a data port are multiplexed together and inputted through the same input port, with every third byte belonging to the same packet. The timing in which the bytes of a packet are inputted will determine which string matching engine is used to search its payload. The process of searching for matching strings in a packet works as follows. The first character or byte being searched is inputted into the string matching block, with a start signal being set to indicate that it is the first character. This character will then retrieve its default transition information from the lookup

table. This default transition information and the character will then be registered by the string matching engine searching the packet payload on its rising clock edge. The state transitioned to will be determined by the default transition information because it is the first character, meaning that it can only transition to a state with a depth of one, or to the start state. Information on the state transitioned to will be requested from the memory used to store the state machine.

The string matching engine will register the next character from the packet it is searching, along with the default transition information that this character will have returned from the lookup table on its next rising clock edge. It will also register the state information that will have been requested from memory on the previous clock cycle. From this information it will then decide whether to traverse to a state pointed to by a transition pointer stored at the state retrieved from memory or to a state pointed to by one of the default transition pointers obtained from the lookup table. This process will continue until the end of the packet is reached. A matching string will have been found if the 12-bit *matching strings* number of a state transitioned to contains an address other than zero. This *matching strings* number is used to indicate if any strings have been matched when a state is entered and if so the location in memory of these matching string numbers. The memory location of the matching string numbers will be sent to the match scheduler along with a set bit.

5.4.3 String Matching Engine Architecture

The architecture of a string matching engine can be seen in Fig. 5.13. Each engine contains registers used to store the current input character, previous two input characters, state information returned from memory, default transition pointer information returned from the lookup table and a register used to store the state type to be analysed. An engine also contains comparator blocks and multiplexers used to analyse the state and default transition pointer information. The first byte from the payload of a packet being searched will be registered to the *Char1* register, while the default transition pointer information it will have retrieved is registered to the *DTP Info* register. The *Start* input signal will be set as this is the first byte from the packet's payload. This means that the *State Address* signal will be set to the address of the state pointed to by the default transition pointer which

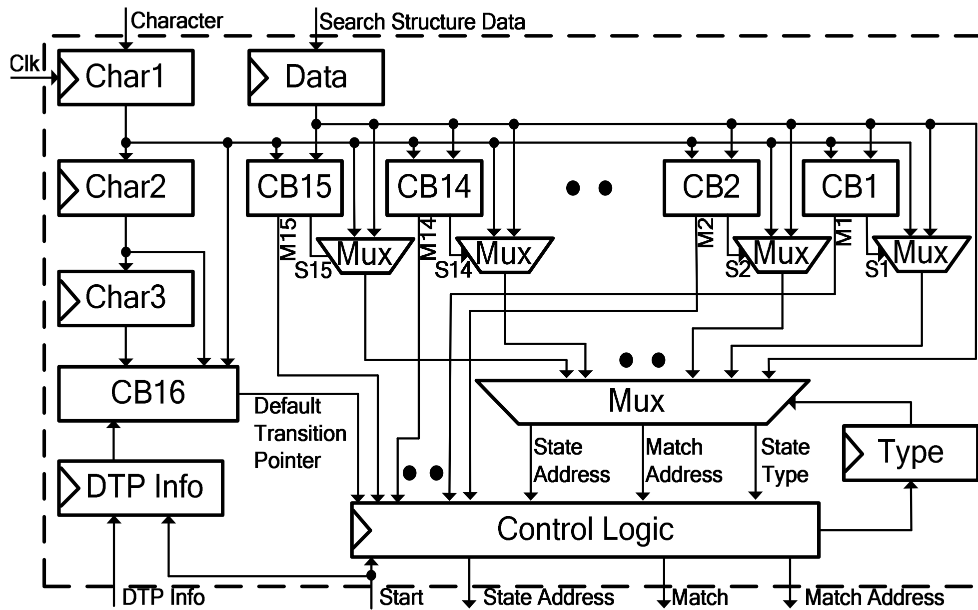


Fig. 5.13. Architecture of the string matching engine.

could be a state with a depth of one or the start state. The states pointed to by all default transition pointers are type 15 states that can store up to 13 transition pointers. This means that the value 15 will be registered to the *Type* register used to record the state type to be analysed on the next clock cycle.

The new input character is registered to the *Char1* register on the next clock cycle, while the *Char2* register records the previous input character. The default transition pointer information that the new input character will have retrieved is registered to the *DTP Info* register, and the information on the state returned from memory is registered to the *Data* register. This state information and the new input character are fed into comparator blocks 1-15 and their multiplexers. These comparator blocks and their multiplexers are used to analyse the different state types. Each comparator block consists of comparators used to compare the input character to a state's transition pointers to see if any are valid. A comparator block will output a set *match* signal and the number of the transition pointer if the input character matches one of the state's transition pointers. The number of the transition pointer is inputted into the multiplexer associated with the comparator block and used to select the appropriate address and type of the state pointed to.

A different comparator block and multiplexer is used for each state type because they contain different numbers of transition pointers and their information is

stored at different positions in a memory word. Due to their simplicity the logic required to use a different comparator block and multiplexer for each state type is less than the amount of logic it would take to shift and parse data, so that a single comparator block and multiplexer could be used. The type and memory location of the state to be traversed to are inputted from the 15 multiplexers to another multiplexer, where the state type information stored in the *Type* register on the previous clock cycle is used to select the correct data.

This multiplexer also selects the correct matching strings information on the current state. This information will be passed to the string matching scheduler shown in Fig. 5.12. This information notifies the string matching scheduler if strings have been matched and if so the memory location of the matching string numbers. The *match* signal from comparator blocks 1-15 are analysed to see if a valid transition pointer has been found at the state returned from memory. A set *match* signal will mean setting the *state address* signal to that of the valid transition pointer in order to retrieve the state it points to from memory. It also means that the state type of the state pointed to can be stored to the *Type* register. No valid transition pointer being found will mean looking at the default transition pointers for the current input character. The previous input character recorded by the *Char2* register is used to check if any of the four default transition pointers to states at a depth of two should be used. None of these being valid will mean using the default transition pointer that points to a state at a depth of one or the start state. Whichever default transition pointer is used will mean setting the *state address* signal so that the state pointed to by the default transition pointer will be retrieved from memory. The value of the *Type* register will also be set to 15.

Finally, on the third and subsequent clock cycles the new input character is registered to the *Char1* register, the *Char2* register will record the previous input character, while the *Char3* register records the input character previous to that. The default transition pointer information that the new input character will have retrieved is registered to the *DTP Info* register and the information on the state returned from memory is registered to the *Data* register. The steps explained will be repeated again, with the exception that the default transition pointer which points to states at a depth of three can now be considered.

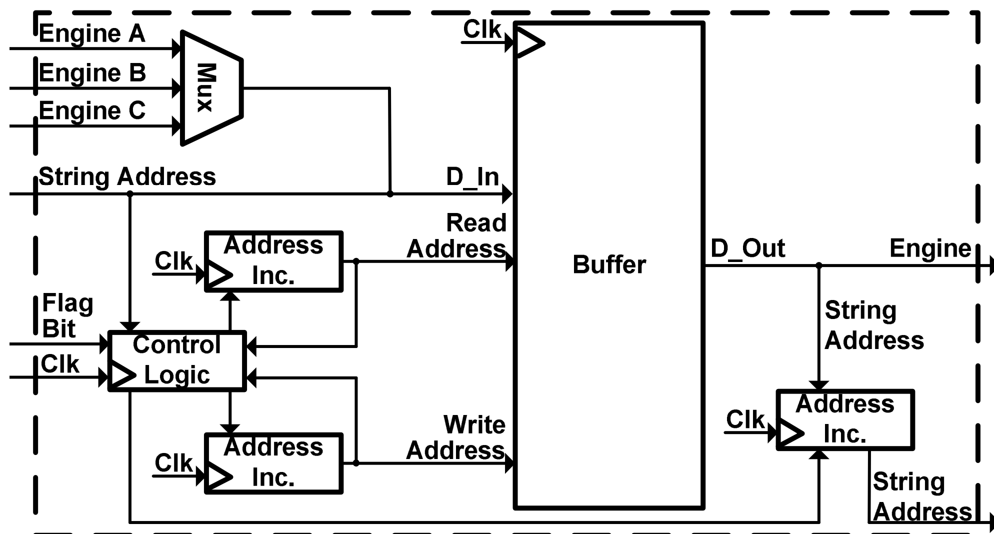


Fig. 5.14. Architecture of the string matching scheduler.

5.4.4 String Matching Scheduler Architecture

The final block to be explained is the string matching scheduler, which can be seen in Fig. 5.14. The string matching scheduler is used to prevent a reduction in throughput when retrieving the numbers of the matched strings from memory during the searching of a packet's payload. The scheduler is shared by three string matching engines. An engine will notify the scheduler that it has found strings being sought. It will also give their location in memory and then leave the scheduler to retrieve the matching string numbers. The scheduler will record the number of the engine that found the strings and the memory location of the matching string numbers in a buffer. The engine that found the strings is recorded as it is used to identify which packet contained the matching strings. The scheduler uses the *Address Inc.* logic block to increment the buffer's write address once this information has been stored.

The number of the engine that recorded the matched strings will be outputted from the hardware accelerator once it reaches the front of the buffer. The memory location of the matching string numbers will also be used to retrieve the matching string numbers from memory. These numbers are outputted two at a time from memory, with a single matching string meaning that one of these numbers is zero. The memory will return a *Flag Bit* to the scheduler to notify it if all matching string numbers have been outputted. This *Flag Bit* not being set will mean using the *Address Inc.* logic block to increment the address of the matching string

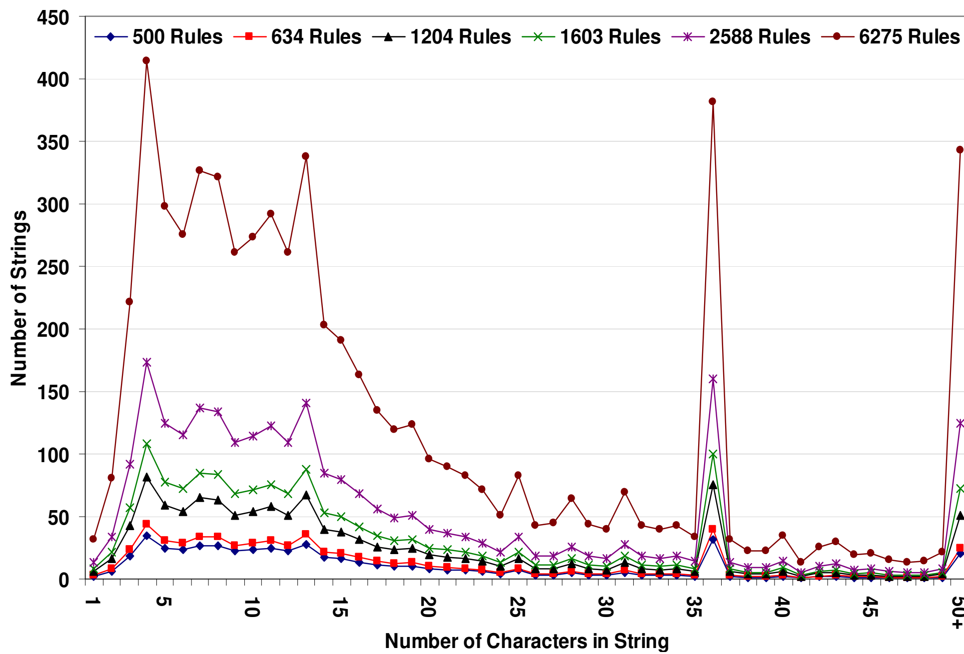


Fig. 5.15. Distribution of string lengths for unique strings found in Snort ruleset.

numbers so that the next two can be outputted. This process continues until a set *Flag Bit* is returned. A set bit being returned will cause the buffer's read address to be incremented using the *Address Inc.* logic block, allowing the reading of the information stored in the next buffer slot.

5.5 Performance Results

5.5.1 Characteristics of Snort Ruleset Used in Testing

The strings used to test the algorithm and hardware accelerator were taken from the Snort 2.6.0 ruleset explained in Section 2.4.1. This ruleset contains 6,275 unique strings that need to be searched for, with the average number of characters contained within a string being 22.65. The length distribution of these strings can be seen in Fig. 5.15. It shows that there is a peak in the number of strings containing between 4 and 13 characters, with the longest string containing 364 characters. The large number of strings, combined with a wide variation in string lengths, shows that string matching methods should be avoided that have a run-time proportional to the number of strings or their length. The algorithm and hardware accelerator presented here can guarantee a fixed throughput irrespective of the number of strings or their length. The distribution of the string lengths for

Table 5.1. FPGA resource utilisation for string matching hardware accelerators.

Device	Logic element usage	Memory usage	f_{max}
Cyclone III	35,511/119,088 (30%)	M9Ks 404/432 (94%)	233.15 MHz
Stratix III	69,585/254,400 (27%)	M9Ks 822/864, M144Ks 0/48 (50%)	460.19 MHz

five smaller rulesets that were created from the Snort ruleset can also be seen in Fig. 5.15. These smaller rulesets were created to test the performance of the algorithm and hardware accelerator in terms of memory usage when searching for different amounts of strings. The strings in these rulesets were chosen using a program created that deletes strings from the Snort ruleset until only a user defined amount remains. The program deletes these strings while trying to match the string length distribution of the Snort ruleset as closely as possible.

5.5.2 Hardware Implementation Parameters

The hardware accelerator has been implemented in VHDL and targeted two devices:

- A Cyclone EP3C120F484C7 FPGA, which is built on TSMC 65nm process technology, running at 1.2 Volts.
- A Stratix EP3SE260H780C2 FPGA, which is also built on TSMC 65nm process technology, running at 1.1 Volts.

The Stratix III implementation has been implemented with six string matching blocks, with each block using 3,584 324-bit memory words to store its state machine and 2,048 27-bit memory words to store the matching string numbers. Memory limitations have meant restricting the Cyclone III implementation to four string matching blocks, with each using 2,560 324-bit memory words to store its state machine and 2,048 27-bit memory words to store the matching string numbers. The architectures were synthesised using Altera Quartus II design software to obtain maximum clock speeds and resource utilisation statistics. Table 5.1 shows the memory and logic usage for the hardware accelerators, along with the maximum clock speed of their memory.

It can be seen that the maximum obtainable clock speed of the Cyclone III memory is 233.15 MHz when it is used to implement the hardware accelerator. Each string matching engine in a string matching block runs at one third the clock

speed of memory, meaning that it can search through each byte of a packet's payload at a guaranteed rate of 77.72 million bytes per second (0.33×233.15 MHz), giving it a maximum throughput of 621.73 Mbps (77.72×8 bits). The memory used in a string matching block is dual port, giving it enough bandwidth to support six string matching engines, putting the maximum throughput for a string matching block at 3.73 Gbps (6×621.73 Mbps). This will also be the hardware accelerator's maximum throughput when searching for strings contained within very large rulesets. This is because the strings will need to be broken up and saved across the memory of all four string matching blocks. The string matching blocks will therefore need to work together, with each block searching for a subset of the strings in a packet's payload.

The throughput of the hardware accelerator will increase to 7.46 Gbps when the strings being searched for only need to be broken up into two groups, with the search structure for each group placed in a separate string matching block. The hardware accelerator will be able to use two pairs of string matching blocks, with each pair capable of searching for all strings in a packet's payload. Each pair will have a throughput of 3.73 Gbps. A maximum throughput of 14.92 Gbps is possible for rulesets whose search structure is small enough to fit in the memory of a single string matching block as a packet will only need to use one block to search its payload for all strings. The throughput will therefore be equal to the sum of all four blocks.

The maximum obtainable clock speed of the Stratix III memory is 460.19 MHz when it is used to implement the hardware accelerator. Each of its string matching blocks will therefore be able to process packets at a speed of 7.36 Gbps. The Stratix III implementation has six string matching blocks, which means that strings can be left as a single group or split into groups of two, three or six. Strings split into groups of six will have the lowest throughput of 7.36 Gbps as all six blocks are required to search through a packet's payload. This throughput increases to 14.73 when strings are split into three groups and saved across three blocks, 22.09 Gbps when two blocks are used to store the strings needed to search a packet's payload and a maximum throughput of 44.18 Gbps when a single block can be used to search a packet's payload.

Table 5.2. Reduction in number of transition pointers stored in states.

Strings	634	1603	2588	6275	500	1204	2588
Aho-Corasick							
States	11,796	29,155	46,301	109,467	9,329	22,026	46,301
Avg.Pointers	68.29	81.07	85.00	87.01	67.28	77.07	85.00
New Method	**Stratix III implementation**				**Cyclone III implementation**		
Blocks	1	2	3	6	1	2	4
States	11,796	29,226	46,599	109,638	9,329	22,049	46,570
d1	68	97	108	110	67	83	125
Avg.Pointers	8.16	6.77	5.33	4.16	7.17	5.70	5.28
d1+d2	262	493	662	1,131	246	415	723
Avg.Pointers	3.43	2.68	2.09	1.92	2.87	2.21	2.20
d1+d2+d3	323	622	850	1,509	306	531	955
Avg.Pointers	2.39	2.01	1.9	1.54	2.09	1.88	1.18
Reduction	96.5%	97.5%	97.8%	98.2%	96.9%	97.6%	98.6%
Mem.(bytes)	148,259	296,967	445,641	838,298	105,599	214,141	429,656
Speed(Gbps)	44.18	22.09	14.73	7.36	14.92	7.46	3.73

5.5.3 Transition Pointer Reduction

The results in Table 5.2 show the reduction that can be achieved in the average number of transition pointers that need to be stored at a state and thus the memory consumption for the Snort ruleset. This reduction is highlighted by showing the average number of transition pointers that need to be stored at a state for both the Aho-Corasick algorithm and the new algorithm presented. It also shows the throughput for the Cyclone III and Stratix III implementations of the hardware accelerator when searching for different numbers of strings. An explanation of the reduction in transition pointers and throughput for the rulesets containing 634 and 6,275 strings is given for the Stratix III implementation to aid understanding.

It can be seen that the average number of transition pointers that need to be stored at a state is 68.29 when using the Aho-Corasick algorithm to build a state machine for the ruleset containing 634 strings. This ruleset contains strings with 68 unique starting characters. This means that there will be 68 states at a depth of one in the state machine. Inserting default transitions to these states in a lookup table reduces the average number of transition pointers that need to be stored in a state to 8.16. Further reductions are achieved by inserting default transition pointers to the four most commonly pointed to states at a depth of two for each ASCII character. This

will bring the average number of transition pointers that a state will need to store down to 3.43 and the total number of default transition pointers stored in the lookup table to 262. The average number of transition pointers that need to be stored at a state decreases to 2.39 when default transition pointers are inserted in the lookup table to the most commonly pointed to state at a depth of three for each ASCII character. This brings the total number of default transition pointers in the lookup table to 323 and reduces the average number of transition pointers that need to be stored in a state by 96.5% when compared to the Aho-Corasick algorithm. The memory required for storing the entire lookup table, state machine and matching string numbers is 148,259 bytes for the 634 strings used. A string matching block will therefore have enough memory to store the total search structure in the Stratix III implementation, enabling the hardware accelerator to achieve its peak throughput of 44.2 Gbps. This is because all six blocks can work separately, searching a packet's payload by themselves.

The average number of transition pointers that a state will need to store is 87.01 when the Aho-Corasick algorithm is used to build a state machine for the Snort ruleset containing 6,275 strings. The memory required to store the search structure for this ruleset is too large to fit in a single string matching block. It therefore has to be split into six separate groups and saved across the six string matching blocks. A total of 110 default transition pointers to states at a depth of one are needed for the six resulting state machines. This will bring the average number of transition pointers that need to be stored at a state down from 87.01 to 4.16. These six state machines will require a total of 1,021 default transition pointers to point to the four most commonly pointed to states at a depth of two for each ASCII character. This will reduce the average number of transition pointers stored at a state to 1.92. The average number can be further reduced to 1.54 by using default transition pointers to states at a depth of three. The resulting search structure needs a total of 838,298 bytes to save the lookup tables, state machines and matching string numbers for the six search structures. The hardware accelerator will have a total throughput of 7.36 Gbps, with all six string matching blocks being needed to search a packet's payload.

It can be seen that the memory consumption scales very well as the number of strings grow when using the new algorithm and hardware accelerator. The number

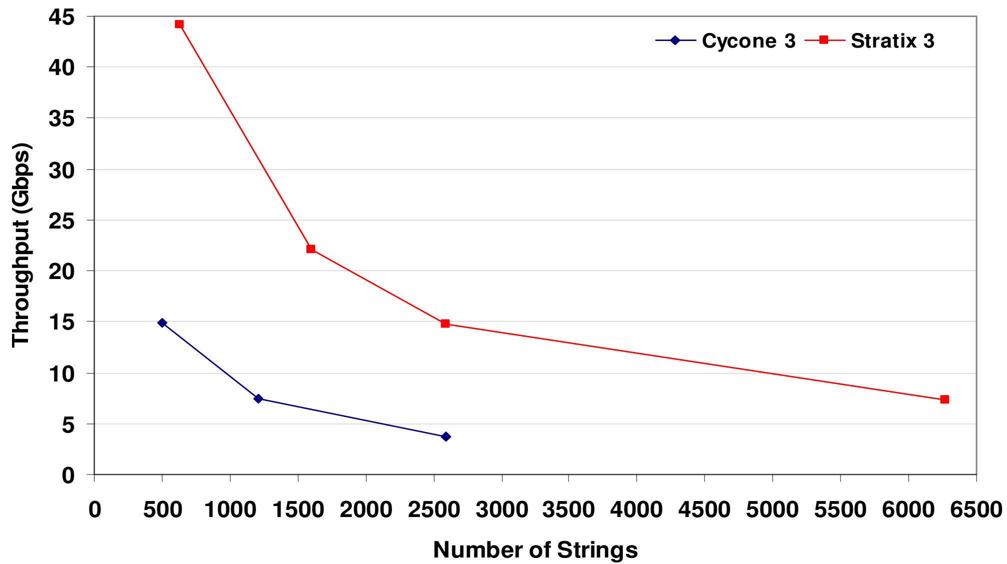


Fig. 5.16. Throughput of the string matchers when using different sized rulesets.

of bits needed to store each string actually decreases as the number of strings increase. This is because the hardware accelerator allows the strings to be broken up into multiple groups, with the state machine for each group placed in a separate string matching block.

Fig. 5.16 shows the achievable throughput for the two implementations of the hardware accelerator when compared to the number of strings being sought using the rulesets shown in Fig. 5.15. It can be seen that the Stratix III implementation performs better than the Cyclone III implementation. This is because it has the largest amount of memory available, allowing it to employ the most string matching blocks. It also has the highest maximum clock speed of the two FPGAs. The Stratix III implementation is able to reach speeds of over 40 Gbps, meaning that it is ideally suited to being deployed at the core of a network. The Cyclone III implementation would work better at the edge of a network as its maximum speed is 14.92 Gbps.

It is worth noting that only half of the Stratix III memory is used. The use of the other half of this memory and some extra logic would allow the Stratix III implementation to support twice as many string matching blocks. This would double the hardware accelerator's throughput when searching for the strings contained within the rulesets used for testing. This is because there would be twice as many blocks available to search through the payload of the incoming

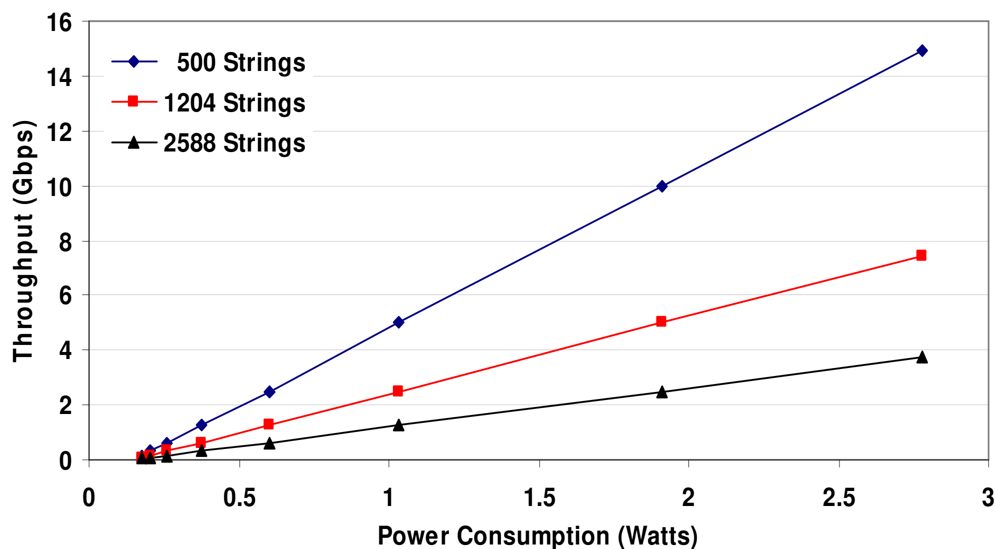


Fig. 5.17. Power consumed by Cyclone III implementation of the string matcher.

packets. The hardware accelerator could also be used to search for twice as many strings, as the strings could be split into twice as many groups, with the state machine for each group saved in a separate block.

5.5.4 Throughput vs. Power Consumption

Post place and route simulations were carried out using the Quartus II PowerPlay Power Analyzer Tool to analyse VCD files generated by ModelSim. These simulations were carried out to measure the power consumed by the hardware accelerator when implemented on the two FPGAs. Fig. 5.17 shows the power consumed by the Cyclone III implementation when configured to process traffic at different levels of throughput for the different sized rulesets used in testing. This graph was created by measuring the hardware accelerator's power consumption, while its clock speed and traffic volume were adjusted to different levels of throughput. It can be seen that the Cyclone III implementation has a maximum power consumption of 2.78 Watts when all four string matching blocks are operating at their highest obtainable clock speed. The three sets of strings used in testing will have different throughputs ranging from 3.73 to 14.92 Gbps at this peak power consumption because they require a different number of string matching blocks to search the payload of a packet.

Fig. 5.18 shows the power consumed by the Stratix III implementation when configured to process traffic at different levels of throughput for the four rulesets

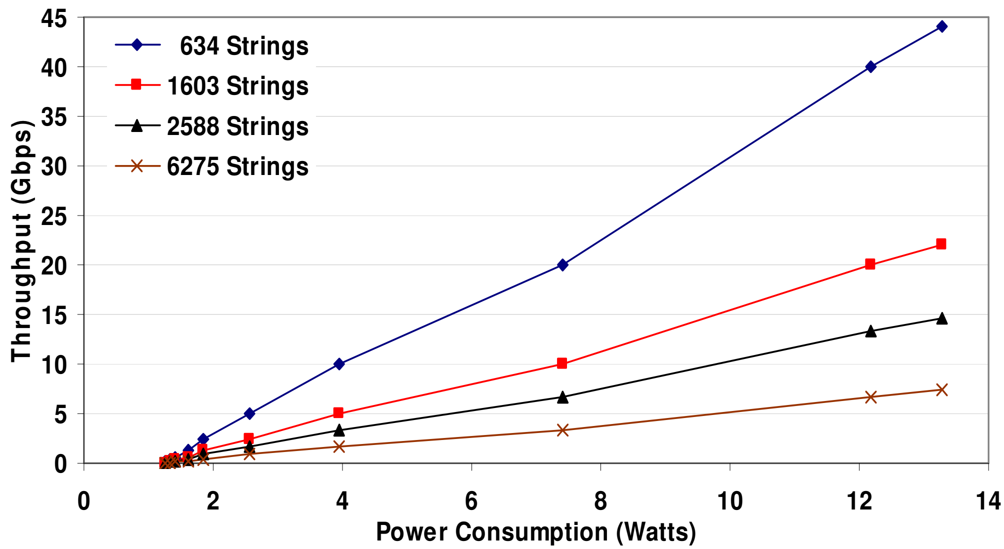


Fig. 5.18. Power consumed by Stratix III implementation of the string matcher.

used in testing. It can be seen that it has a peak power consumption of 13.28 Watts when its six string matching blocks are running at their maximum clock speed. Like the Cyclone III implementation it also has different levels of throughput, ranging from 7.36 to 44.18 Gbps at its peak power consumption. This is again due to the different number of string matching blocks required to match a packet's payload to the strings contained within the different sized rulesets. As mentioned in Section 5.5.3, it is possible to double the throughput or amount of strings that can be searched for by doubling the number of string matching blocks. This would, however, cause a large increase in the power consumption due to extra switching and the activation of extra sections of the FPGA. The Stratix III implementation uses almost five times as much power as the Cyclone III implementation. This is because the Stratix III is a much larger device, consuming more static power. It also operates at a much higher clock speed, resulting in higher amounts of dynamic power consumption.

5.5.5 Evaluation Against Prior Art

This section compares the new string matching algorithm and hardware accelerator to the work in [54], which presents two string matching algorithms and their hardware implementations. All approaches are state machine-based, with their performance compared in terms of throughput and amount of memory required to save the search structure needed to locate strings in the payload of a

Table 5.3. Performance comparison of string matching hardware accelerators.

Approach	Device	Memory (bytes)	Throughput (Gbps)
New method	Cyclone III	138,470	7.46
New method	Stratix III	138,470	22.09
Bitmap[54]	ASIC	2,800,000	7.8
Path compression [54]	ASIC	1,100,000	7.8

packet. These results can be seen in Table 5.3. The schemes presented in [54] use bitmaps to reduce the amount of memory needed to save a state's transition pointers and path compression to reduce the number of states that need to be saved. These schemes were tested using rules from an older Snort ruleset that contained 19,124 characters. The Snort ruleset used here with 6,275 rules contains 142,129 characters, so for fair comparison the program explained in Section 5.5.1 was used to reduce the number of strings, while still keeping the same string length distribution until only 19,124 characters were left.

It can be seen that the new algorithm presented here requires 20 times less memory to save the total data structure needed to search for strings when compared to the scheme that uses bitmap compression. The new algorithm presented also shows a reduction in memory consumption when compared to the scheme that uses path compression, requiring eight times less memory. A direct comparison on throughput is not easy as the bitmap and path compressed schemes were simulated running on an ASIC using 130nm process technology while the hardware accelerator presented here was implemented using FPGAs that are built using 65nm process technology. It would, however, be safe to assume that a hardware accelerator implemented as an ASIC using 130nm technology would perform equal to and if not better than a hardware accelerator implemented on an FPGA using 65nm technology.

Looking at Table 5.3, it can be seen that the Cyclone III implementation shows equal performance to the bitmap and path compressed schemes in terms of throughput, while the Stratix III implementation performs three times better. This performance increase can be attributed to the fact that the algorithm presented here does not use fail pointers, while the other two schemes do. The use of fail pointers means that there will be wasted transitions when traversing the decision tree and a worst case throughput cannot be guaranteed. Also there is a large logic

delay associated with bitmaps as finding the correct transition pointer involves the checking and addition of the 256 bits contained within the bitmap. The hardware accelerator presented here only requires a comparison of no more than thirteen 8-bit ASCII characters which can be carried out in parallel.

5.6 Summary of Contributions

This chapter has shown that it is possible to implement the computationally heavy task of string matching at the line speed of a backbone network, with low power consumption. A new algorithm is presented that uses a state machine with eliminated fail pointers to guarantee worst case performance. The algorithm uses a small number of default transition pointers to the most commonly pointed to states in the state machine. These default transition pointers are placed in a lookup table where they are shared by all states in the state machine, greatly reducing the number of pointers that must be stored at a state. This allows the search structure created for rulesets containing thousands of strings to be compact enough so that it can be easily packed into the on-chip memory of an FPGA.

The chapter also introduces a hardware accelerator architecture that implements the algorithm and employs multiple string matching engines. These engines can be configured to work together, searching a single packet when a very large ruleset is used. They can also be configured to work separately, searching multiple packets in parallel when a smaller ruleset is used, thus achieving maximum throughput. The new string matching algorithm and hardware accelerator architecture also show large improvements in throughput and memory consumption when compared to other hardware-based approaches.

Chapter 6 – Conclusions and Future Work

6.1 Conclusions

This section summarises the research objectives of this thesis and the results achieved by the work described in previous chapters.

6.1.1 Motivation for Proposed Research – A Summary

The large plethora of services being provided by ISPs and the growing number of sophisticated attacks on networks that need to be blocked have made the tasks of packet classification and Deep Packet Inspection (DPI) increasingly difficult. Packet classification is required to perform services such as traffic billing based on Internet usage, network security, giving priority to VoIP and IPTV packets, rate limiting, load balancing and resource reservation. It does this by matching a packet's header to a set of rules, with the rule matched determining the flow a packet belongs to and all packets in a particular flow being processed in a similar manner. The increasing number of services that need to be provided means that the number of rules used to separate incoming packets into appropriate flows has grown from hundreds to thousands of rules. An important part of DPI is fixed string matching. Fixed string matching is used to search for strings in a packet's payload that are associated with known attacks. The number of strings that need to be searched for to detect attacks can be several thousand if rulesets from popular network intrusion prevention and detection systems such as Snort are used.

The constant growth in Internet usage has further complicated the tasks of packet classification and fixed string matching, with classifiers being required to classify up to 125 Mpps and fixed string matching hardware accelerators given only 0.2 ns to search through each byte of a packet's payload at 40 Gbps line speeds. Another

challenge in implementing these tasks is the tight power budget on a router's line card which determines that any new hardware used to process packets must be energy efficient to reduce operating costs and prevent power related problems such as overheating.

6.1.2 Summary of Thesis Contributions

The work presented in this thesis tackles the problems associated with packet classification and fixed string matching by presenting new algorithms and hardware accelerators that prevent them from becoming a packet processing bottleneck if implemented at the core of a network. The algorithms build search structures that use low amounts of memory when compared to existing algorithms. They are also tailored towards hardware implementation, allowing for ultra-high throughput. The hardware accelerators presented use low power memories such as SRAM rather than power hungry TCAM, which is commonly used in networking applications. The contributions made are summarised in the following three sections.

6.1.3 Packet Classification

An extensive analysis of popular packet classification algorithms was carried out in Chapter 2 comparing their performance in terms of memory usage, power consumption and throughput when operating on a processor similar to the type used as a processing core in programmable network processors. This analysis showed HyperCuts to be one of the best all-round performers, scaling well when rulesets containing thousands of rules are used to classify packets. Chapter 3 presented hardware accelerators that implement modified versions of the HyperCuts packet classification algorithm. HyperCuts is a decision tree-based algorithm that divides the hyperspace of a ruleset into multiple groups so that each group contains only a small number of rules that are suitable for a linear search. The algorithm was modified so that no floating point division is required when traversing the decision tree to find the group of rules that must be searched. This is done to reduce the complexity of the hardware accelerator's logic, increasing clock speeds and throughput. Floating point division was removed by replacing the region compaction heuristic used by HyperCuts to reduce a decision tree's

memory consumption with a new heuristic that uses pre-cutting. Pre-cutting also reduces memory consumption while only requiring simple shift and AND operations to be performed when traversing the decision tree. The cutting scheme was also modified to make the algorithm better suited to using the wider memory words employed by the hardware accelerators presented. Modifications were also made to how rules are stored through simple encoding schemes that improve the storage efficiency of rulesets.

One of the hardware accelerator architectures presented in Chapter 3 uses ultra-wide memory words and is capable of classifying up to 169 Mpps when using rulesets containing up to 49,000 rules. It has been designed to cope with problem rulesets that contain many wildcard rules. Rulesets that contain wildcard rules are difficult to break into small groups suitable for a short linear search. The use of ultra-wide memory words gives the hardware accelerator the ability to access the information required to search up to 48 rules in a single clock cycle. This means that it can quickly find a matching rule when searching the large group of rules found in decision trees built from rulesets containing many wildcard rules. The chapter also presents two other packet classification hardware accelerators that use reduced width memory words. The use of reduced width memory makes these hardware accelerators better suited to classifying packets when using rulesets that do not contain a large number of wildcard rules. This is because they can only access enough information to search two rules per clock cycle which means that it must be possible to break the rulesets being used into groups where each group contains a small number of rules.

One of the hardware accelerators that uses reduced width memory has been designed to use on-chip memory while the other has been designed to use external memory. The architecture that uses on-chip memory can classify up to 433 Mpps when using rulesets that contain up to 80,000 rules. The architecture that uses external memory is capable of classifying packets when using rulesets containing up to a million rules. All packet classification hardware accelerators use multiple classification engines. This gives them the ability to break problem rulesets containing a large number of wildcard rules into groups, with a separate packet classification engine used to search the decision tree built for each group. This can

help to reduce the worst case number of clock cycles needed to classify a packet and lower memory consumption. The hardware accelerators have been compared to state of the art packet classifiers that use TCAM, with results showing an increase in throughput of up to 325% and a decrease in power consumption of up to 81%.

6.1.4 Frequency Scaling

Another contribution made towards the field of packet classification is an adaptive clocking unit that is presented in Chapter 4. It dynamically adjusts the clock speed to a packet classifier so that its available processing capacity matches the processing needs of the network traffic on a router's line card. This is done to keep power consumption low at times when a network's traffic volume is light. The adaptive clocking unit stores the headers of the incoming packets in a small buffer and uses the number of packets stored to decide the clock frequency of the packet classifier. A scheme was developed to keep clock frequencies at the lowest speed capable of servicing the line card while keeping frequency switches to a minimum. A low power architecture for packet classification was implemented as an ASIC and using FPGAs. It consisted of the adaptive clocking unit and the packet classification engine presented in Chapter 3 that uses ultra-wide memory words. The low power architecture was tested extensively using synthetic 2.5 Gbps, 10 Gbps and 40 Gbps packet traces created from real network traces obtained from the NLANR database while classifying packets using synthetic rulesets containing up to 25,000 rules. Simulation results show that power savings of between 14-88% can be made when the adaptive clocking unit is used rather than a fixed clock speed.

6.1.5 String Matching

A new multi-pattern matching algorithm and hardware accelerator are presented in Chapter 5 that are used to carry out fixed string matching. They can search through a packet's payload at a guaranteed rate of one character per clock cycle no matter how many strings are being sought or the length of these strings. This prevents attackers from being able to flood the system by constructing packet payloads that the fixed string matcher performs poorly on. The new algorithm is a

modified version of the Aho-Corasick algorithm that builds a state machine from the strings being sought. The largest cause of memory consumption in such a state machine is the transition pointers stored at each state. Transition pointers are used to select the state that should be transitioned to on any given clock cycle, with the input characters used to select the appropriate transition pointer that must be followed. The new algorithm stores transition pointers to the most commonly pointed to states in a small lookup table. These transition pointers are called default transition pointers and they are shared by all states in the state machine. This reduces memory consumption by over 98% when compared to the original Aho-Corasick algorithm.

The hardware accelerator that implements the new algorithm can search for thousands of strings at speeds of over 40 Gbps which is fast enough to meet core network line speeds. It uses multiple string matching blocks that can be configured to work together, searching a single packet when a very large ruleset is used. They can also be configured to work separately, searching multiple packets in parallel when a smaller ruleset is used, allowing maximum throughput to be achieved. It has been tested extensively using the Snort ruleset which contains 6,275 unique strings that must be searched for. A comparison with other state of the art string matching hardware accelerators and algorithms show that the algorithm and hardware accelerator presented here can reduce memory consumption by over 87% while increasing throughput by 283%.

6.2 Future Work

The fixed string matching algorithm and hardware accelerator presented in Chapter 5 help to provide the processing capacity necessary to carry out the computationally heavy task of DPI at the core of a network, where line speeds can reach up to 40 Gbps. DPI will still, however, remain a packet processing bottleneck until algorithms and hardware accelerators are provided that make it possible for multi-match packet classification and regular expression matching to be performed at the core of a network. A logical progression for the work carried out in this thesis would be to modify the algorithms and hardware accelerators presented so that they can perform multi-match packet classification and regular expression matching. Another progression for the work carried out would be to

design a power saving mechanism capable of dynamically adjusting the processing capacity of the fixed string matching hardware accelerator so that its processing capacity matches the processing needs of the network traffic. This would allow for a reduction in the amount of dynamic power used. The following three sections explain briefly how this future work could be carried out.

6.2.1 Multi-Match Packet Classification

The packet classification hardware accelerator presented in Chapter 3 that uses reduced width memory words could easily be modified so that it returns all matching rules rather than only the matching rule with the highest priority. This could be done by always searching a leaf node until its end is reached and outputting all matched rules found along the way. This would not increase the worst case number of memory accesses required to classify a packet, making it possible for a modified version of the hardware accelerator to perform multi-match packet classification at speeds of up to 138.56 Gbps. The architecture of the modified hardware accelerator could even be made simpler than the architecture presented in Chapter 3. This is because it would not need to compare matching results between engines in order to find the matching rule with the highest priority in the case where a ruleset has been split into multiple groups with a separate packet classification engine used to search each group. This is due to the fact that all matching rule IDs will be outputted rather than just the ID of the matching rule with the highest priority.

6.2.2 Regular Expression Matching

Deterministic Finite Automata (DFA) is commonly used to carry out the task of regular expression matching [30, 31, 32, 34, 35]. The hardware accelerator presented in Chapter 5 also uses DFA to implement fixed string matching. It would, however, need some modifications in order to make it better suited to implementing regular expression matching. It currently uses default transition pointers to states near the start state. These transition pointers are shared by all states, leading to large memory reductions when carrying out fixed string matching. This is because fixed string matching does not allow the use of wildcard characters. It is also because the content being searched for varies widely

between strings. This results in a state machine where the transition pointers at most states will typically only point to the same few states near the start state. Regular expression matching allows the use of wildcard characters, which results in a state machine where a state's transition pointers will tend to point deeper into the state machine. This problem could be overcome by using extra default transition pointers to states further away from the start state. The use of wildcard characters also means that states tend to store more transition pointers. The hardware accelerator would also need to be modified so that states can store more transition pointers to allow for this. The algorithm used to build the state machine would also need to be modified so that it can handle regular expressions and be able to intelligently select the default transition pointers that will lead to the largest memory savings.

6.2.3 Reducing the Fixed String Matching Hardware Accelerator's Power

Finally, the fixed string matching hardware accelerator presented in Chapter 5 requires six string matching blocks when implemented on an FPGA to meet core network line speeds of 40 Gbps. The processing capacity of these string matching blocks will not be fully utilised at times of low traffic volume, resulting in unnecessary dynamic power being used. The use of multiple processing elements makes this hardware accelerator ideally suited to clock gating, where the clock to unneeded processing elements is gated at times of low traffic volume, reducing dynamic power consumption. A scheme similar to the one used in Chapter 4 could be employed to decide how many processing elements are needed to cope with the processing needs of the incoming traffic. This would involve employing a small buffer to capture the incoming bytes of a packet's payload and using the number of bytes stored to decide how many processing elements should be active. The same methods used in Chapter 4 to keep frequency switches to a minimum could also be used to reduce the number of times the clocks to processing elements are gated in order to reduce the processing delays associated with the activating and deactivating of processing elements.

APPENDIX A – POWER USAGE

The following figures show the average power consumed by the ASIC, Cyclone III and Stratix III implementations of the low power packet classifier, when they are used to classify packets from 2.5 Gbps, 10 Gbps and 40 Gbps traces, using search structures built for the ACL, FW and IPC rulesets containing 5,000 and 25,000 rules.

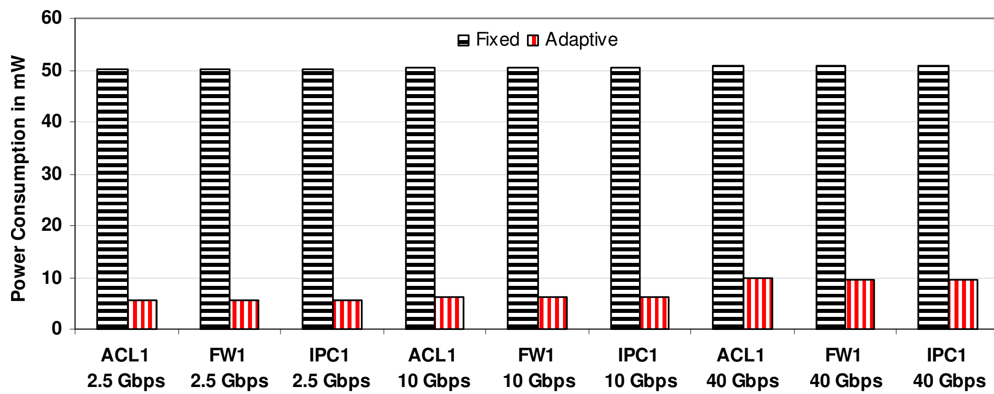


Fig. A. 1. Power usage of ASIC low power classifier using 5,000 rules.

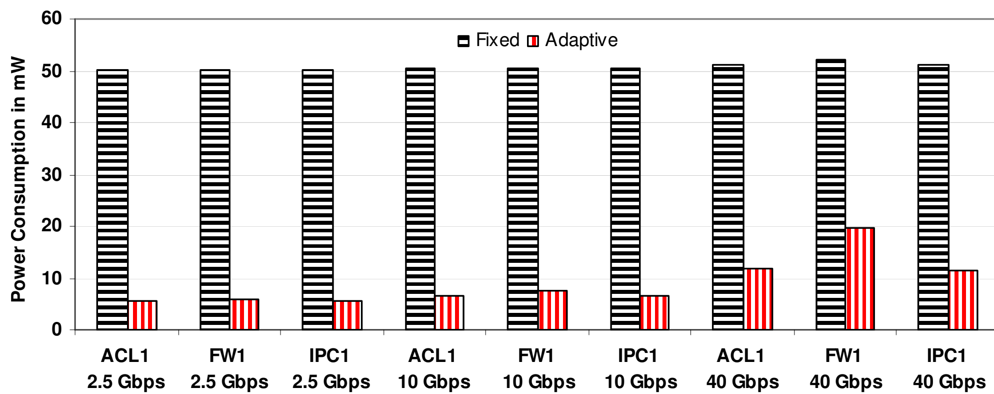


Fig. A. 2. Power usage of ASIC low power classifier using 25,000 rules.

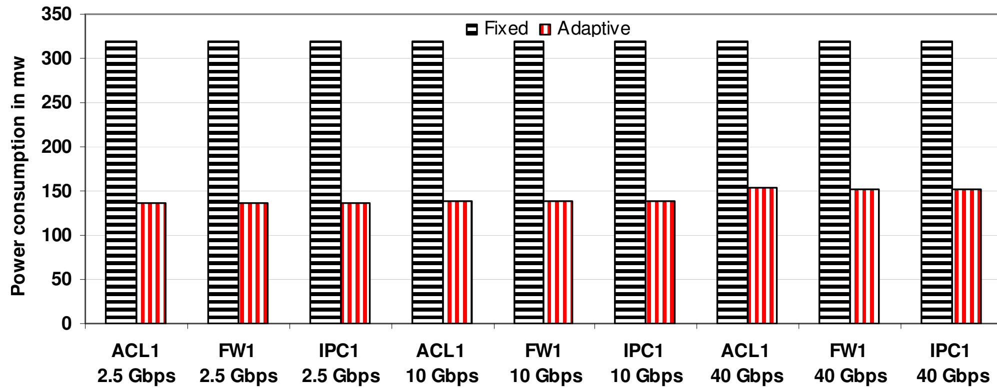


Fig. A. 3. Power usage of Cyclone III low power classifier using 5,000 rules.

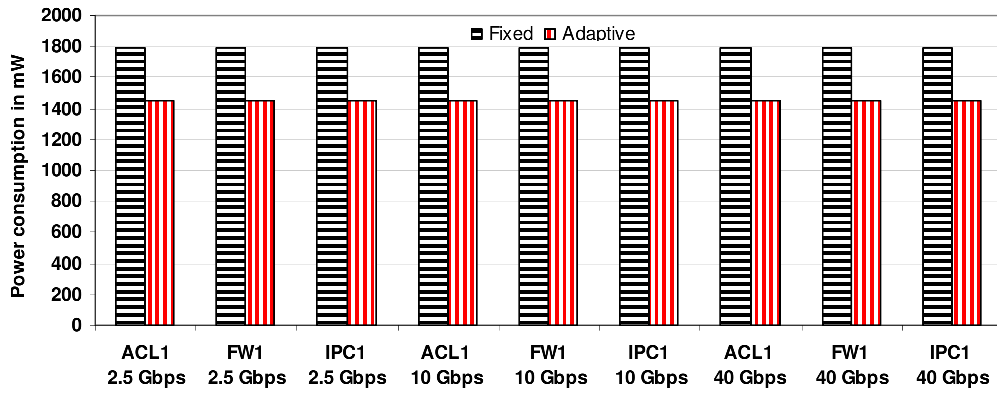


Fig. A. 4. Power usage of Stratix III low power classifier using 5,000 rules.

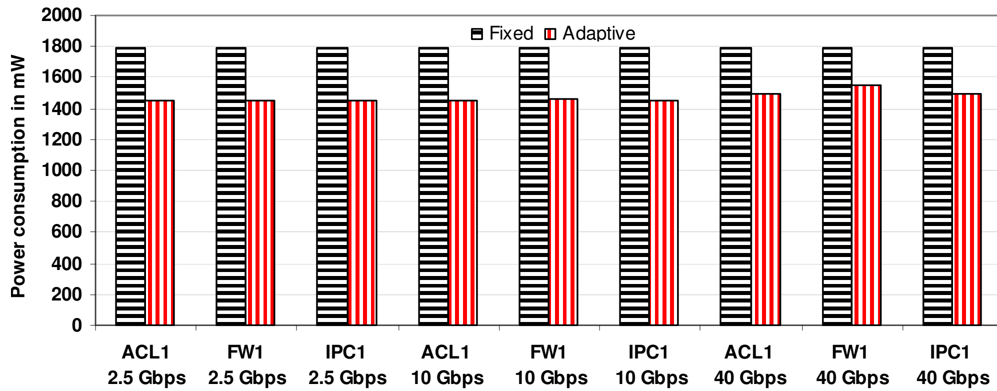


Fig. A. 5. Power usage of Stratix III low power classifier using 25,000 rules.

BIBLIOGRAPHY

[1] Internet World Stats, Usage and Population Statistics. [Online]. Available: http://www.internetworldstats.com/stats.htm

[2] M. Gupta and S. Singh, “Greening of the Internet, ”*In Proc. ACM SIGCOMM*, (Aug. 2003), pp. 19-26.

[3] A. Gallo, “Meeting Traffic Demands with Next-Generation Internet Infrastructure,” *Lightwave*, vol. 18, no. 5, (May 2001), pp.118–123.

[4] Cisco ASR 9000 Series Aggregation Services Router. [Online]. Available: http://www.cisco.com/en/US/docs/routers/asr9000/hardware/ethernet_line_card/installation/guide/asr9kELCIGapaspecs.html

[5] N. Shah, “Understanding network processors,” *Tech. Rep. Version 1.0*, (Sept. 2001).

[6] Intel IXP2800 Network Processor Product brief. [Online]. Available: http://download.intel.com/design/network/ProdBrf/27905403.pdf

[7] H. Zimmermann, “OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection,” *IEEE Trans. on Communications*, vol. 28, no. 4 , (April 1980), pp. 425-432.

[8] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Trans. Comput. Syst*, vol. 2, no. 4, (Nov. 1984), pp. 277-288.

[9] P. Gupta and N. McKeown, “Packet classification using hierarchical intelligent cuttings,” *IEEE Micro*, vol. 20, no. 1, (Feb. 2000), pp. 34-41.

[10] S. Singh, F. Baboescu, G. Varghese and J. Wang, "Packet Classification Using Multidimensional Cutting," *In Proc. ACM SIGCOMM*, (Aug. 2003), pp. 213-224.

[11] M. Abdelghani, S. Sezer, E. Garcia and M. Jun, "Packet Classification Using Adaptive Rules Cutting (ARC)," *In Proc. of the Advanced industrial Conference on Telecommunications/Service Assurance with Partial and intermittent Resources Conference/E-Learning on Telecommunications Workshop*, (July 2005), pp. 28-33.

[12] P. Gupta and N. McKeown, "Packet classification on multiple fields," *In Proc. ACM SIGCOMM*, (Sep. 1999), pp. 147-160.

[13] T.V. Lakshman and D. Stiliadis, "High-Speed Policy based Packet Forwarding Using Efficient Multi-dimensional Range Matching", *In Proc. ACM SIGCOMM*, (Sep. 1998), pp. 203-214.

[14] F. Baboescu and G. Varghese, "Scalable packet classification," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, (Feb. 2005) pp. 2-14, 2005.

[15] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?," *In Proc. IEEE INFOCOM*, (April 2003) , pp. 53-63.

[16] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification using Tuple Space Search," *In Proc. ACM SIGCOMM*, (Sep. 1999), pp. 135-146.

[17] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network Mag.*, vol. 15, no. 2, (Mar. 2001), pp.24-32.

[18] T. Woo, "A modular approach to packet classification: algorithms and results," *In Proc. IEEE INFOCOM*, (Mar. 2000), pp. 1213-1222.

[19] P. C. Wang, C. T. Chan, C. L. Lee and H. Y. Chang "Scalable Packet Classification for Enabling Internet Differentiated Services," *IEEE Trans. on Multimedia*, vol. 8, no. 6, (Dec. 2006), pp. 1239-1249.

[20] Cypress Ayama 10000 Network Search Engine. [Online]. Available: http://download.cypress.com.edgesuite.net/design_resources/datasheets/contents/cynse10256_8.pdf

[21] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *In Proc. IEEE Security and Privacy*, vol. 1, no. 4, (Jul. 2003), pp. 33-39.

[22] D. Moore, C. Shannon, and J. Brown, "Code-Red: A Case Study on The Spread and Victims of an Internet Worm," *In Proc. of the 2nd ACM Internet Measurement Workshop*, (Nov. 2002), pp. 273–284.

[23] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," *In Proc. of the 13th USENIX conference on System administration*, (Nov. 1999), pp. 229-238

[24] S. Antonatos, K. G. Anagnostakis and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," *In Proc. of the 4th international Workshop on Software and Performance*, (Jan. 2004), pp. 207-215.

[25] F. Yu, R. H. Katz and T. V. Lakshman, "Efficient Multimatch Packet Classification and Lookup with TCAM," *In IEEE Micro*, vol. 25, no. 1 (Jan. 2005), pp. 50-59.

[26] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," *In Proc. of the ACM/SIGDA 13th international Symposium on Field-Programmable Gate Arrays,* (Feb. 2005), pp. 238-245.

[27] M. Nourani and M. Faezipour, "A Single-Cycle Multi-Match Packet Classification Engine Using TCAMs," *In Proc. of the 14th IEEE Symposium on High-Performance Interconnects*, (Aug. 2006), pp. 73-80.

[28] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," *In Proc. of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (May 2001), pp. 227-238.

[29] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," *In Proc. of 13th International Conference on Field Programmable Logic and Applications*, (Sep. 2003), pp. 956-959.

[30] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” *In Proc. ACM SIGCOMM*, (Sep. 2006), pp. 339-350.

[31] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman and R. H. Katz, “Fast and memory-efficient regular expression matching for deep packet inspection,” *In Proc. of the 2nd ACM/IEEE Symposium on Architecture For Networking and Communications Systems*, (December 2006), pp. 93-102.

[32] S. Kuma, J. Turner and J. Williams, “Advanced algorithms for fast and scalable deep packet inspection,” *In Proc. of the 2nd ACM/IEEE Symposium on Architecture For Networking and Communications Systems*, (Dec. 2006), pp. 81-92.

[33] I. Sourdis, J. Bispo, J. M. P. Cardoso and S. Vassiliadis, “Regular Expression Matching in Reconfigurable Hardware,” *In Journal of Signal Processing Systems*, vol. 51, no. 1, (Oct. 2007), pp. 99-121.

[34] M. Becchi, and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,” *In Proc. of the 3rd ACM/IEEE Symposium on Architecture For Networking and Communications Systems,* (Dec. 2007), pp. 145-154.

[35] S. Kumar, B. Chandrasekaran, J. Turner and G. Varghese, “Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia,” *In Proc. of the 3rd ACM/IEEE Symposium on Architecture For Networking and Communications Systems*, (Dec. 2007), pp. 155-164.

[36] D.E. Taylor and J.S. Turner, “ClassBench: a packet classification benchmark,” *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, (June 2007), pp. 499-511.

[37] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Commun. ACM*, vol. 18, no. 6, (Jun. 1975), pp. 333-340.

[38] Sim-Panalyzer, The SimpleScalar-ARM Power Modeling Project. [Online]. Available: http://www.eecs.umich.edu/~panalyzer/

[39] Evaluation of Packet Classification Algorithms. [Online]. Available: http://www.arl.wustl.edu/~hs1/PClassEval.html

[40] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, (Dec. 2009), pp.2435–2463.

[41] Cisco IOS IPS Signature Deployment Guide. [Online]. Available: http://www.cisco.com/

[42] J. Levandoski, E. Sommer, and M. Strait, "Application Layer Packet Classifier for Linux," [Online]. Available: http://l7-filter.sourceforge.net/

[43] SNORT network intrusion prevention and detection system. [Online]. Available: http://www.snort.org

[44] K. Salah and A. Kahtani, "Performance evaluation comparison of Snort NIDS under Linux and Windows Server," *In Journal of Network and Computer Applications*, (Aug. 2009)

[45] D.E. Knuth, J.H. Morris and V.R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, (June 1977), pp. 323-350.

[46] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, (Oct. 1977), pp. 762-772.

[47] B. Commentz-Walter, "A string matching algorithm fast on the average," *In Proc. 6th International Colloquium on Automata, Languages, and Programming*, (July 1979), pp. 118-132.

[48] J. J. Fan and K. Y. Su, "An efficient algorithm for matching multiple patterns," *IEEE Trans. on Knowledge and Data Engineering*, vol. 5, no. 2, (April 1993) pp. 339-351.

[49] U. Manber and S. Wu, "A fast algorithm for multi-pattern searching," *In Tech.Report TR-94-17*, 1994.

[50] M. Fish and G. Verghese, "Fast content-based packet handling for intrusion detection," *In UCSD Technical Report CS2001-0670*, 2001.

[51] M. Crochemore and D. Perrin, "Two-way string-matching," *J. ACM*, vol. 38, no. 3, (Jul. 1991), pp. 650-674.

[52] J. V. Lunteren "High-Performance Pattern-Matching for Intrusion Detection," *In Proc. IEEE INFOCOM*, (April 2006), pp. 1-13.

[53] B. Soewito, L. Vespa, A. Mahajan, N. Weng, and H. Wang, "Self-addressable memory-based FSM: a scalable intrusion detection engine," *IEEE Network Magazine*, vol. 23 no. 1 (Jan 2009), pp. 14-21.

[54] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection." *In Proc. IEEE INFOCOM*, (Mar. 2004), pp. 333-340.

[55] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *In Proc. of the 32nd Annual international Symposium on Computer Architecture,* (June 2005), pp. 112-122.

[56] S. Dharmapurikar, P. Krishnamurthy, T. Sproull and J. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro*, vol. 24, no. 1, (Jan. 2004), pp. 52-61.

[57] Titan-IC Systems Parallel String Matcher. [Online]. Available: http:// www.titanicsystems.com/products/item/2/parallel-string-matcher-psm/

[58] FPGA solutions from Xilinx. [Online]. Available: http://www.xilinx.com/

[59] FPGA solutions from Altera. [Online]. Available: http://www.altera.com/

[60] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," *In Proc. of the 11th IEEE international Conference on Network Protocols*, (Nov. 2003), pp. 120-131.

[61] K. Zheng, H. Che, Z. Wang, B. Liu and X. Zhang, "DPPC-RE: TCAM-Based Distributed Parallel Packet Classification with Range Encoding," *IEEE Trans. on Computers*, vol. 55, no. 8, (Aug. 2006), pp. 947-961.

[62] D. Pao, Y. K. Li and P. Zhou, "An encoding scheme for TCAM-based packet classification," *In Proc. of the 8th International Conference on Advanced Communication Technology*, (Feb. 2006), pp. 470-475.

[63] F. Yu, R. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM." *In Proc. of the 12th IEEE international Conference on Network Protocols*, (Oct. 2004), pp. 174-183.

[64] M. Alicherry, M. Muthuprasanna, V. Kumar, "High speed matching for network IDS/IPS", *In Proc. of the Proceedings of the 2006 IEEE international Conference on Network Protocols*, (Nov. 2006), pp.187-196.

[65] J. Sung, S. Kang, Y. Lee, T. Kwon, and B. Kim, "A Multi-gigabit Rate Deep Packet Inspection Algorithm using TCAM," *In Proc. IEEE Globecom*, (Nov. 2005), pp. 453-457.

[66] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for content filtering," *In Proc. of the 1st ACM Symposium on Architecture for Networking and Communications Systems*, (Oct. 2005), pp. 183-192.

[67] G. K. Yeap, *Practical Low Power Digital VLSI Design*, 1st ed. Dordrecht The Netherlands: Kluwer Academic Publishers, 1998.

[68] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2nd ed. Prentice Hall, 2003.

[69] H. J. Veendrick, "Short-Circuit Dissipation of Static CMOS Circuitry and its Impact on the Design of Buffer Circuits", *Journal of Solid-State Circuits*, vol. 19, no. 4, (Aug. 1984) pp. 468-473.

[70] A. P. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*, 1st ed. Dordrecht The Netherlands: Kluwer Academic Publishers, 1995.

[71] A. Kinane, "*Energy Efficient Hardware Acceleration of Multimedia Processing Tools*," Ph.D. dissertation, School of Electronic Engineering, Dublin City University, (Apr. 2006), [Online]. Available: http://elm.eeng.dcu.ie/~kinanea/thesis/kinane_final.pdf

[72] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, (Dec. 2003), pp. 68–75.

[73] J. A. Butts and G. S. Sohi, “A Static Power Model for Architects,” *In Proc. Of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, (Dec. 2000), pp. 191–201.

[74] L. Wei, Z. Chen, K. Roy, M. C. Johnson, Y. Ye, and V. K. De, “Design and optimization of dual-threshold circuits for low-voltage low-power applications,” *IEEE Trans. on Very Large Scale Integration Systems*, vol. 7, no. 1, (Mar. 1999), pp. 16–24.

[75] D. A. Pucknell and K. Eshragian, *Basic VLSI Design*, 3rd ed. Australia: Prentice Hall, 1994.

[76] TSMC 65nm Technology Platform, Taiwan Semiconductor Manufacturing Company. [Online]. Available: http://www.tsmc.com

[77] W. Ruby, (Low) Power To The People, *EDAVision Magazine*, (Mar. 2002)

[78] A. Krishnamoorthy, (July 2004), Minimize IC Power Without Sacrificing Performance, [Online]. Available: http://www.eedesign.com/article/showArticle.jhtml?articleId=23901143

[79] F. Poppen. (2002, May), Low Power Design Guide. [Online]. Available: http://www.lowpower.de/charter/lpdesignguide.pdf

[80] A. P. Chandrakasan, S. Sheng and R. W. Brodersen, “Low-power CMOS digital design,” *Journal of Solid-State Circuits,* vol. 27, no. 4, (Apr. 1992), pp. 473-484.

[81] H. Song, and J. W Lockwood, “Efficient packet classification for network intrusion detection using FPGA,” *In Proc. of the 2005 ACM/SIGDA 13th international Symposium on Field-Programmable Gate Arrays*, (Feb. 2005), pp. 238-245.

[82] K. Lakshminarayanan, A. Rangarajan and S. Venkatachary, “Algorithms for advanced packet classification with ternary CAMs,” *In Proc. of the 2005 Conference on Applications, Technologies, Architectures*, and Protocols For Computer Communications, (Aug. 2005), pp. 193-204.

[83] J. V. Lunteren and T. Engbersen, “Fast and scalable packet classification,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, (May 2003), pp. 560–571.

[84] D. Shah and P. Gupta, “Fast incremental updates on Ternary-CAMs for routing lookups and packet classification,” *In Proc. Hot Interconnects*, (Aug. 2000), pp. 145–153.

[85] K. Zheng, H. Che, Z. Wang, and Bin Liu, “TCAM-based Distributed Parallel Packet Classification Algorithm with Range-Matching Solution”, *In Proc. IEEE INFOCOM*, (Mar. 2005), pp. 293-303.

[86] S. Dharmapurikar, H. Song, J. Turner and J. Lockwood, “Fast packet classification using bloom filters,” *In Proc. of the 2nd ACM/IEEE Symposium on Architecture For Networking and Communications Systems*, (Dec. 2006), pp. 61-70.

[87] D.E. Taylor and J.S. Turner, “Scalable packet classification using distributed crossproducting of field labels,” *In Proc. IEEE INFOCOM*, (Mar. 2005), pp. 269-280.

[88] W. Jiang and V. K. Prasanna, “Large-scale wire-speed packet classification on FPGAs,” *In Proc. of the ACM/SIGDA international Symposium on Field Programmable Gate Arrays*, (Feb. 2009), pp. 219-218.

[89] AT&T Completes Next-Generation IP/MPLS Backbone Network, World's Largest Deployment of 40-Gigabit Connectivity [Online]. Available:http://www.att.com/gen/press-room?cdvn=news&newsarticleid=26230&pid=4800

[90] Passive Measurement and Analysis Project, National Laboratory for Applied Network Research. [Online]. Available: http://pma.nlanr.net

[91] Corporation for Education Network Initiatives in California trace. [Online]. Available: ftp://pma.nlanr.net/traces/long/cnic/1/

[92] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot, “Packet-level traffic measurements from the sprint IP backbone,” *IEEE Network*, vol. 17, no. 6, (Nov.-Dec. 2003), pp. 6–16.

[93] A. Dainotti, A. Pescape, and G. Ventre, “A Packet-level Characterization of Network Traffic,” *In proc. of the 11th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks*, (Jun. 2006), pp.38–45.

[94] T.Mudge, “Power: A First-Class Architectural Design Constraint,” *Computer*, vol. 34, no. 4, (Apr. 2001), pp.52–58.

[95] K. Chun and A. Ling, (2003, Nov. 24), Placement Approach Cuts SoC Power Needs. [Online]. Available: http://www.eetimes.com/in focus/silicon engineering/OEG20031121S0035.

[96] R. Bhutada and Y. Manoli, “Complex clock gating with integrated clock gating logic cell,” *In Proc. of the - 2007 International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, (Sep. 2007), pp. 164-169.

[97] L. Hai, S. Bhunia, Y. Chen, T.N. Vijaykumar and K. Roy, “Deterministic clock gating for microprocessor power reduction,” *In Proc. of the 9th International Symposium on High-Performance Computer Architecture*, (Feb. 2003), pp. 113-122.

[98] Y. Luo, J. Yu, J. Yang and L. Bhuyan “Low power network processor design using clock gating,” *In Proc. of the 42nd Annual Design Automation Conference*, (June 2005), pp. 712-715.

[99] Y. Luo, J. Yu, J. Yang, and L. N. Bhuyan, “Conserving network processor power consumption by exploiting traffic variability”, *ACM Trans. on Architecture and Code Optimization*, vol. 4, no. 1 (Mar. 2007)

[100] R. Kokku, U. B. Shevade, N. S. Shah, M. Dahlin and H. M. Vin “Energy-Efficient Packet Processing”, *University of Texas at Austin Technical Report TR04-04.*

[101] A. Mallik, B. Lin, G. Memik, P. Dinda and R.P. Dick, “User-Driven Frequency Scaling,” *IEEE Computer Architecture Letters*, Vol. 5, no. 2, (Feb. 2006), pp. 61-64.

[102] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas and M.L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," *In Proc. of the 18th International Symposium on High-Performance Computer Architecture*, (Feb. 2002), pp. 29-40.

[103] A. Chattopadhyay and Z. Zilic, "GALDS: a complete framework for designing multiclock ASICs and SoCs," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 13, no. 6, (Jun. 2005), pp. 641–654.

[104] K.J. Nowka, G.D. Carpenter, E.W. MacDonald, H.C. Ngo, B.C. Brock, K.I. Ishii, T.Y. Nguyen, and J.L. Burns, "A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, (Nov. 2002), pp. 1441-1447,

[105] P. Pillai K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," *In Proc. of the 18th ACM Symposium on Operating Systems Principles*, (Oct.2001). pp. 89-102.

[106] T. Pering, T. Burd and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," *In Proc. of International Symposium on Low Power Electronics and Design*, (1998), pp. 76-81.

[107] K. Usami and M. Horowitz, "Clustered voltage scaling technique for low-power design," *In Proc. of the 1995 international Symposium on Low Power Design*, (Apr. 1995), pp. 3-8.

[108] Y. Luo, J. Yang, L. Bhuyan and L. Zhao, "NePSim: A Network Processor Simulator with Power Evaluation Framework", *IEEE Micro Special Issue on Network Processors for Future High-End Systems and Applications*, vol. 24, no. 5, (Oct. 2004), pp. 34-44.

[109] C. T. Chow, L. S. M. Tsui, P. H. W. Leong, W. Luk, and S. Wilton, "Dynamic voltage scaling for commercial FPGAs," *IEEE International Conference on Field Programmable Technology*, (Dec. 2005), pp.173-180.