BRANCH PREDICTION FOR NETWORK PROCESSORS

by

David Bermingham, B.Eng

Submitted in partial fulfilment of the requirements

for the Degree of Doctor of Philosophy

School of School of Electronic Engineering

Supervisors: Dr. Xiaojun Wang

Prof. Lingling Sun

September 2010

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: _____
David Bermingham (Candidate)

ID: _____

Date: _____

TABLE OF CONTENTS

ABSTRACT

Originally designed to favour flexibility over packet processing performance, the future of the programmable network processor is challenged by the need to meet both increasing line rate as well as providing additional processing capabilities. To meet these requirements, trends within networking research has tended to focus on techniques such as offloading computation intensive tasks to dedicated hardware logic or through increased parallelism. While parallelism retains flexibility, challenges such as load-balancing limit its scope. On the other hand, hardware offloading allows complex algorithms to be implemented at high speed but sacrifice flexibility. To this end, the work in this thesis is focused on a more fundamental aspect of a network processor, the data-plane processing engine. Performing both system modelling and analysis of packet processing functions; the goal of this thesis is to identify and extract salient information regarding the performance of multi-processor workloads. Following on from a traditional software based analysis of programme workloads, we develop a method of modelling and analysing hardware accelerators when applied to network processors. Using this quantitative information, this thesis proposes an architecture which allows deeply pipelined micro-architectures to be implemented on the data-plane while reducing the branch penalty associated with these architectures.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AES Advanced Encryption Standard

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

ATM Asynchronous Transfer Mode

CAM Content Addressable Memory

CISC Complex Instruction Set Computer

CMOS Complementary Metal Oxide Silicon

CPI Cycles Per Instruction

CPU Central Processing Unit

DPI Deep Packet Inspection

DRAM Dynamic Random Access Memory

FIFO First-In-First-Out

GHR Global History Register

GPP General Purpose Processor

HTTP HyperText Transfer Protocol

IC Integrated Circuit

ILP Instruction Level Parallelism

IP Internet Protocol

IPsec Internet Protocol Security

IPTV Internet Protocol Television

ISA Instruction Set Architecture

ISP Internet Service Providers

MAC Media Access Controller

NAT Network Address Translation

NIDS Network Intrusion Detection System

NP Network Processor

OS Operating System

OSI Open Systems Interconnection

PCI Peripheral Component Interconnect

PE Process Engine

PHT Pattern History Table

POS Packet Over SONET

QoS Quality of Service

RISC Reduced Instruction Set Computer

RTP Real Time Protocol

RTSP Real Time Streaming Protocol

SHA Secure Hash Algorithm

SoC System on Chip

SRAM Static Random Access Memory

SSL Secure Socket Layer

TCAM Ternary Content Addressable Memory

TCP Transmission Control Protocol

TSL Transport Layer Security

UDP User Datagram Protocol

VOIP Voice Over IP

VPN Virtual Private Network

WAN Wide Area Network

WLAN Wireless Local Area Network

LIST OF PEER-REVIEWED PUBLICATIONS

D. Bermingham, Z. Liu, X. Wang, B. Liu., "Field-Based Branch Prediction for Packet Processing Engines", *The Fifteenth International Conference on Parallel and Distributed Systems (ICPADS'09)*, December 2009.

D. Bermingham, Z. Liu, X. Wang and B. Liu., "Branch Prediction For Network Processors", *ICM 2008, Proceeding of the International Conference on Microelectronics*, December 2008.

D. Bermingham, Z. Liu and X. Wang., "SimNP:A Flexible Simulation Framework for Network Processor Systems", (Short Paper) *ANCS '08 : Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, November 2008.

Z. Liu, D. Bermingham and X. Wang., "Towards Fast and Flexible Simulation of Network Processors", *CIICT2008 : Proceeding of the IET 2008 China-Ireland International Conference on Information and Communications Technologies*, August 2008.

D. Bermingham, A. Kennedy, X. Wang, B. Liu., "A Survey of Network Processor Workloads", *CIICT2007 : Proceeding of the IET 2007 China-Ireland International Conference on Information and Communications Technologies*, August 2007.

A. Kennedy, D. Bermingham, X. Wang, B. Liu., "Power Analysis of Packet Classification Algorithms", *ICSPC 2007 : Proceedings of the 2007 IEEE International Conference on Signal Processing and Communications*, December 2007.

D. Bermingham, X. Wang, and B. Liu., "Analysis of FPGA-Based AES Round Architectures", *ISSC 2005 : Proceeding of the 2005 IEE Irish Signals and Systems Conference*, June 2006.

X. Zhang, B. Liu, W. Li, Y. Xi, D. Bermingham and X. Wang., "IPv6-oriented 4OC768 Packet Classification Scheme with Deriving-Merging Partition and Field-variable Encoding Algorithm", *IEEE INFOCOM2006*, 23-29 April 2006.

ACKNOWLEDGEMENTS

I would like to thank the people who have helped my undertaking of this thesis. Firstly, my thesis supervisor Xiaojun Wang for all his advice, guidance and patience throughout the years I have been part of the Network Processing Group in DCU. Network processors was a new topic to both of us when I began my thesis and so I am particularly grateful to Xiao for allowing me the time to find a topic which interested me. I would also like to thank Liu Zhen, a past member of the NPG. Zhen provided invaluable help with almost all aspects of my work towards the end of my research. I would like to thank Jim Dowling who gave me the opportunity to undertake a Ph.D. in DCU.

I am extremely grateful to my family who have always provided me with endless support and encouragement, especially my mother who has always taught me to value education and my father who has a boundless optimism.

Finally, I would like to thank Jeanie for her love, patience and sacrifices throughout my time as a Ph.D student. I am not the most organised of individual at the best of times and without her 'work ethic' I doubt this would have ever been completed. This work is as much due to her as me.

CHAPTER 1

Introduction

1.1 Network Processors

The Internet has grown to encompass the global networking system underlying modern business, research, communication and collaboration. Developments in network technologies and fibre optics have allowed products and services to be developed on one continent, manufactured on another and serviced on yet another. The challenge of network research is to expand these communications systems to allow more complex services at ever faster speeds. To maximise the packet processing rate it is possible to implement routing using highly tailored Application Specific Integrated Circuits (ASICs), but when application flexibility is required a programmable platform such as a Network Processor (NP) is required. Instead of utilising dedicated hardware logic for packet processing, a NP system is organised around a software-based processing array, comprised of a number of heterogeneous processors (Process Engine (PE)). With the ability to reprogram each PE, software-based routers allow new functions to be deployed on existing network hardware in order to meet changing network demands. Changes to Internet-based protocols, services and applications no longer require extensive hardware changes. For research within the network processor domain the challenge is retaining this flexibility while meeting the needs of both additional processing requirements and ever increasing bandwidth demands. This dissertation addresses some of the difficulties associated with these trends, propos-

ing methods for analysing and improving the performance of the PEs within a network processor.

1.2 Trends Within Networks

In this section some of the trends within network research are outlined, before an examination of the effect of these trends on the future for programmable routers.

1.2.1 Bandwidth Growth

During the 1990's, developments within optical and switching technology, a boom in computer usage in both the home and the office and the dot com bubble resulted in bandwidth doubling on average every year ([1], [2] and [3]). Although growth in capacity slowed down significantly at the beginning of the century, the under-utilised network infrastructure was quickly absorbed and the work in [4] found that between 2003 and 2008 average internet traffic increased by 50% to 60% per annum, with predictions for similar growth over the next three years [5]. This capacity has been utilised by two factors. Firstly, the number of systems connected has increased substantially. Originally limited to universities, governments and large corporations, Internet access is now almost universal in developed countries, with developing countries such as India and China rapidly expanding domestic coverage. For example, the number of people in China with access to the internet has increased from $\sim$50 million users in 2002 to over 250 million users in 2008 [6] but encompasses less than 20% of China's population.

In addition to the total number of networked terminals, both home and corporate network users have demanded faster connections, requiring large scale broadband networks to be deployed. With an increasing amount of trade and commerce provided via electronic systems, companies have become dependent on high bandwidth permanent Internet connections. In countries around the world, broadband services such as Fiber-To-The-Home, Cable or Digital Subscriber Line (DSL) are widely available to both homes and offices, while wireless technologies provide broadband in more remote and mobile environments.

1.2.2 Network Technologies

With the ability to accommodate arbitrary communication systems, IP switched networks have grown from wired networks connected over existing circuit switching systems to the broad Wide Area Network (WAN) which encompasses the modern Internet. Traditional circuit switched systems such as telephony networks are gradually being replaced by packet switched equivalents [7]. Similarly, technologies such as Wireless Local Area Network (WLAN), high speed broadband, mobile broadband and gigabit Ethernet have either replaced existing standards or have been designed in such a way as to be accommodated within the existing infrastructure.

1.2.3 Application and Service Demands

Evolving from a system which provided simple services such as file transfer and email, the Internet today supports applications which were deployed decades after the standardisation of the Internet protocols. Some of the more common applications are described below.

1.2.3.1 World Wide Web

The most transparent Internet application, web browsing has evolved from simply rendering text-based information to the provision of interactive services such as e-commerce and content generation. While the original Web largely followed a simple one way dissemination technique (personal webpages), the modern Web involves more cooperative and collaborative interaction, such as social networking or Wikipedia [8]. Modern developments have allowed applications to be offloaded from end terminals to more centralised systems. A hot topic of research ([9],[10],[11],[12]), this trend is commonly referred to by various terms (Virtualisation, Cloud Computing, Software-as-a-Service(SaaS)). The goal of cloud computing is to allow the design and deployment of highly scalable and highly redundant virtualised systems. Instead of end users, e.g. individuals and companies deploying expensive software and hardware on a per-node basis, a centralised system is provided where the required service can be accessed remotely on-demand. For com-

mercial and academic institutions requiring complex and high performance computing resources, cloud computing allows the high capital costs associated with such systems to be replaced with a service charge for access to the external hosting network.

1.2.3.2 Digital Media

Broadband connections, as well as improvements in the technologies associated with media streaming, have allowed vast amounts of digital media to be published on the Internet at little capital cost. Video and music websites have become increasingly popular, with Youtube [13] streaming over 100 million videos per day to users all over the world. Gill et al. found that video streams from Youtube accounted for almost 5% of campus wide network traffic [14]. In addition, interactive entertainment systems such as computer games have become increasingly networked. Entire online environments have been constructed [15], with systems such as Second Life available for entertainment, information and collaborative purposes [16]. In addition to streaming services used to disperse content is the trend towards two way communication systems such as Voice Over IP (VOIP) or video teleconferencing. Utilising similar algorithms to streaming applications it presents a number of additional challenges at a router level since latency, and therefore processing time, is the primary difficulty in maintaining such services.

1.2.3.3 Peer To Peer

The most significant application in terms of bandwidth usage, Peer-to-Peer (P2P) has evolved from small centralised systems to vast distributed networks, allowing large amounts of data to be efficiently transferred across the Internet. Today, the most common P2P system in operation is the BitTorrent protocol. Although it is difficult to characterise the amount of Internet bandwidth absorbed by P2P applications, research in [17],[18] highlighted the sizeable resources utilised by P2P systems. For Internet Service Providers (ISPs), the large volume of data transferred across Peer-to-Peer networks presents two difficulties. Firstly, a small number of customers can monopolise large amounts of available bandwidth, severely reducing the service available to other customers. Secondly,

the liability regarding the transport of copyrighted material across networks is yet to be resolved. From a network provider's perspective, Sen et al. found that despite the large amounts of data transferred within a P2P system, the stability of P2P traffic lends itself to flow level metering and traffic shaping [18]. Shaping techniques, however, require network providers to implement packet classification as a means of detecting P2P traffic, while more finely grained systems which attempt to filter only illegal material require that routers must be both flow and content aware at a packet level.

While the future use of P2P is difficult to ascertain, a number of trends within Internet based services indicate that P2P based protocols will remain over the medium term. For open source software companies, P2P has become a major means of distributing software, with the most common Linux Operating System (OS) versions utilising BitTorrent based distribution. Secondly, for content and media generation companies, P2P networks provide an efficient means of dispersing files outside of real-time demands. For example, the music service Spotify [19] utilises a P2P system to stream encrypted audio between users.

1.2.3.4 Future Demands

Extrapolating from current Internet trends it is possible to speculate on future Internet demands. Firstly, with available bandwidth continually expanding, both personal and commercial users will push additional functions onto networked systems. Ultra-high bandwidth systems deployed at home will allow television and media to be delivered via Internet Protocol (IP) networks, while commercial institutions can utilise services such as external data-centres providing hosting, backup and storage functionality. Services such as virtualisation will allow computing resources such as storage, bandwidth and processing to be viewed as a commodity, available for varying amounts of time to bidders. The separation between P2P systems and centralised media distribution is becoming increasingly blurred, pushing the bandwidth requirements from the company generating the content to the network providers which connect users.

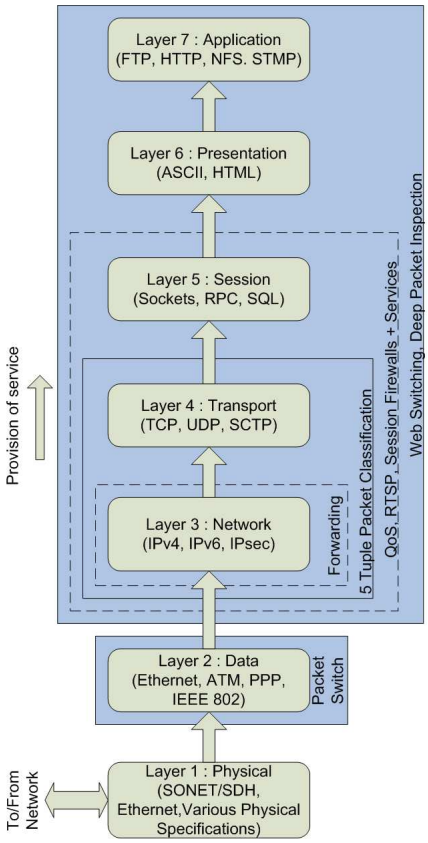


Figure 1.1: 7 Layer OSI Network Model [20]

1.3 Network Trends and Network Processors

While the discussion of network trends outlined in the previous section did not focus on implementation details, the question for network researchers is how these trends affect network designs, topologies and functionality. In the case of a distributed P2P system, the trend has been to push bandwidth demands away from centralised systems to each node of the network. On the other hand, the various aspects of a real-time Internet Protocol Television (IPTV) system, such as Personal Recorders or Video on Demand, rely on a centralised distribution network. For interactive web systems, such as social networks, the data demands are small and not sensitive to security. For commercial operations, which outsource data-processing to external networked sites, the volume of traffic can be very large with security vital to communication. From a network design perspective, the challenge is how these various demands can be met.

In general, these trends have been represented at a router level by an expansion of the type of functions performed on a modern router. Mapped to the traditional Open Systems

Interconnection (OSI) stack [20], the original task of a router has expanded from simple layer three routing and modification to complex application layer functions such as deep packet inspection and intrusion detection. While an Application Specific Integrated Circuit (ASIC)-based router can be developed to provide packet routing or classification functionality, it is difficult to provide a cost-effective solution with many differing demands. A router deployed by local Internet Service Providers (ISP) has little need for packet encryption or intrusion detection, while a router deployed at the edge of a corporate network requires a mechanism for creating a secure Virtual Private Network (VPN) and packet inspection. Quality of Service (QoS) systems require routers to differentiate between 'normal' traffic and latency-sensitive traffic such as IPTV or VOIP. Legal requirements increasingly require an ISP to detect illegal P2P data traversing the network without affecting legal P2P systems.

With Internet based applications constantly evolving, it is increasingly difficult to implement dedicated hardware capable of meeting all demands. While the protocols which underlie the Internet are likely to remain in place for many years to come, the methods, protocols and algorithms for performing applications such as queueing, metering, classification or payload inspection are constantly changing.

1.3.1 The Motivation for This Thesis

With ASIC-based routers lacking flexibility, the programmable NP architecture remains a good alternative to an application-specific router. Unfortunately a replacement NP-based architecture is not without limitations, especially when future requirements are considered. While technological evolution has allowed the performance (e.g. clock frequency, transistors per chip) obtained from digital circuits to greatly increase, it has not been at the growth rate experienced by network bandwidth, creating a performance gap between the growth of bandwidth deployed and the processing capabilities available. Within microprocessor design, the performance increases have been limited by factors such as dynamic power consumption and memory access latency. Multiprocessing and cache hierarchies have allowed performance increases to be obtained through methods other than increases

in microprocessor operating frequency [21].

For an NP architecture, the performance gap between bandwidth and processing speed has commonly been met using two techniques. The first method involves implementing additional processing resources as a means of exploiting flow level parallelism. While each additional PE does allow additional performance to be extracted, it is not a 'for-free' solution and requires a redesign of the memory system, bus hierarchies and load balancing mechanisms. Secondly, the demands for computationally intensive functions such as packet classification, intrusion detection, encryption, etc., have generated scope for extensive research on the topic of hardware accelerators for NP architectures. Instead of implementing applications in software, algorithms are mapped to dedicated hardware blocks. Similar to ASIC based routers, hardware acceleration favours packet processing performance over flexibility. It remains unclear whether hardware acceleration is an efficient solution for all NP applications.

In addition to these two solutions, another method of increasing NP performance involves micro-architectural improvements in the underlying PE architecture. These micro-architectural improvements involve investigating and implementing mechanisms for improving the performance of the PEs. Techniques such as caching, multi-threading, instruction level parallelism (superscalar) or deeper pipelining represent four such mechanisms which can be used to improve PE performance. The focus of this work is primarily on the fourth of these micro-architectural techniques; namely the processor pipeline depth. As with other micro-architectural design techniques, pipelining is not without significant challenges.

To achieve optimum performance within a pipelined design the pipeline must remain full at all times. For a microprocessor, the difficulty arises during conditional operations which attempt to modify program flow. Since the conditional evaluation cannot be known immediately, the processor must either wait while the instruction is evaluated or assume a pre-determined course (typically assuming the branch will not be taken). If the assumed path is incorrect the instructions which are misfetched must be flushed from the pipeline.

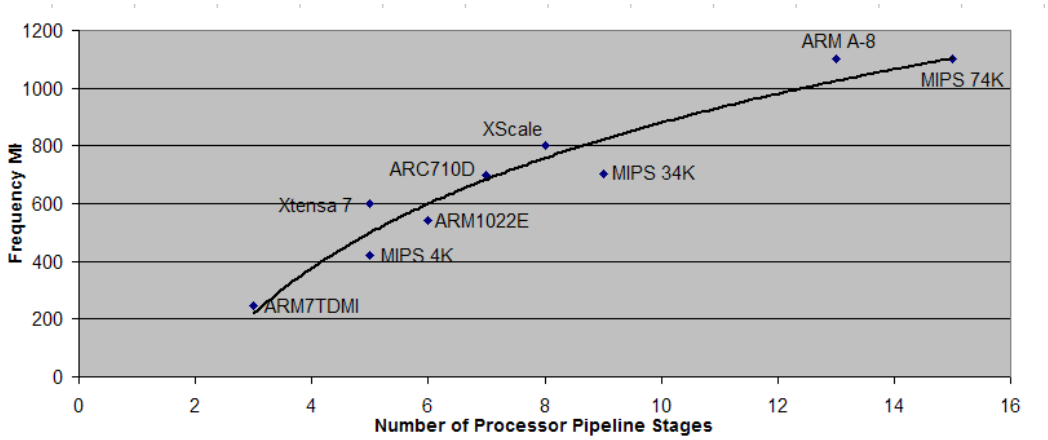


Figure 1.2: Pipeline Depth Vs. Microprocessor Performance

For highly conditional code this branch penalty can become a significant loss of processor cycles and it is possible that a deeper pipeline can actually decrease overall performance. The most common solution to mitigate this problem is to implement a prediction scheme which attempts to *guess* whether a conditional branch will be taken based on previous run-time history.

With NPs requiring an increasing amount of processing capabilities, it is believed that deeper pipelined processors will become one method of increasing performance. Examining commercial Reduced Instruction Set Computer (RISC) processors ([22],[23],[24],[25]) in Figure 1.2, it can be seen that significant increments in operating frequency can be achieved by implementing a deeply pipelined design. For NP research, the goal is how to harness the additional performance available without incurring the penalty associated with conditional operations.

1.4 Research Objectives

The research goals of this thesis can be summarised as follows:

1. To investigate the current state of network processing modelling and simulation as a method for investigating architectural aspects of network processor design.

2. To derive a simulation framework allowing effective benchmarking and analysis of network processor applications.

3. To undertake a comprehensive analysis of network processor workloads, characterising those applications implemented completely in software, and examining the effect of conditional branch instructions in NP applications.

4. To design and implement techniques for mitigating the branch penalty within NP applications as a means of improving network processor performance.

1.5 Thesis Structure

The remainder of this thesis is structured as follows:

- *Chapter 2 – Technical Background*

  The background section introduces the two main topics discussed in this thesis, i.e., programmable network processors and branch prediction for pipelined microprocessors. Topics such as network architectures and applications are briefly covered, highlighting trends within NP design. The second part of this chapter presents a technical introduction to the aspects of microprocessor design which relates to branch prediction, outlining the concepts of pipelining and branch penalties as well as discussing existing methodologies for mitigating the penalty.

- *Chapter 3 – Performance Evaluation Methods for Network Processors*

  Following on from the high level background information and state of the art survey presented in Chapter 2, Chapter 3 presents a more detailed technical background with regards to the topics of performance evaluation of NP and PE architectures. With research such as branch prediction requiring a means of modelling NP platforms, a survey of the methods available to model an NP environment is presented. The second part of this chapter examines the metrics and methodologies by which branch prediction schemes can be examined within an NP environment.

- *Chapter 4 – A New Simulator for Network Processors*

  There is no existing method of efficiently evaluating NP architectures, and so a new simulation framework for NP systems is proposed. Designed to provide the high

turn-around time of a functional simulator, it includes simulation blocks for devices commonly incorporated on an NP System on Chip (SoC).

- *Chapter 5 – Analysis of NP Workloads*

  Using common NP applications and algorithms, an examination and analysis of network tasks is presented. While previous work has focused on complex superscalar designs, this thesis examines more fundamental design considerations, such as parallelism, memory profile and the branch behaviour of NP applications.

- *Chapter 6 – Branch Prediction in Process Engines*

  After identifying branch behaviour as a significant limitation to implementing deeply pipelined PE architectures, Chapter 6 examines branch prediction techniques within an NP environment in detail. Following on from the branch behaviour and analysis presented in Chapter 5, the performance of existing prediction techniques is evaluated before proposing a new flow indexed branch prediction scheme specifically designed for PEs. A complete analysis is provided, with the proposed architecture compared to previous solutions in terms of cost, performance and scalability.

- *Chapter 7 – Conclusions & Future Work*

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

CHAPTER 2

Technical Background

2.1 Overview

The topics discussed in this thesis span two engineering topics, network processors and computer architecture. With this in mind, chapters 2 and 3 present a technical overview of both of these topics. For network processors, this background information provides a context for the current state of the art within NP design and research. An overview of current networking topology (hierarchy, protocol and technology) is first presented before describing how NP platforms fit into the modern Internet. A summary of the current applications deployed on an NP platform is used to highlight some of the challenges to NP development and design. Various methods by which NP performance can be improved are examined in order to outline some of the strengths and disadvantages of each method. One such way by which NP performance could be improved is by increasing the performance of the individual PEs by implementing a deeper pipelined architecture. When compared to other methods of scaling PE, it is argued that implementing a deeper pipelined architecture represents a more efficient method of scaling PE performance, since neither flexibility nor cost is greatly increased.

As was highlighted in Chapter 1 however, deeper micro-processor pipelines are not without possible performance drawbacks. Despite typically increasing the operating frequency of the underlying architecture, the increased number of pipeline stages can actu-

ally result in lower performance if the pipeline cannot be kept full at all times. The latter half of this chapter presents an overview of the concepts of pipelined micro-processors and branch penalties, along with a survey of the existing methods of branch prediction which can be used to mitigate this branch penalty.

2.2 Networks

Within the OSI model (see Figure 1.1)[1], a router has traditionally been defined as a layer 3 device. Unlike a layer 2 Media Access Controller (MAC) or network switch, a router must be aware of the underlying protocol and, for more complex functions, must be aware of higher-level application information. For example, a modern router may be used as a web-switching platform, balancing HyperText Transfer Protocol (HTTP) requests across a number of hosting systems. In general, a network such as the Internet can be divided into three components which are briefly outlined.

2.2.1 Network Protocols

The Internet is comprised of a number of heterogeneous networks communicating within a shared framework. Although many frameworks have been proposed and implemented, the Transmission Control Protocol (TCP) and IP (TCP/IP) protocol suite [26] dominate today's computer networks. The ability to accommodate arbitrary network systems and topologies is derived from the fact that the TCP/IP suite provides no specification regarding hardware layers. Concerned only with transporting data, the TCP/IP suite begins at layer 3 of the OSI model. Today the bulk of traffic found on the Internet is comprised of the layer 3 IP protocol and two layer 4 transport protocols [27] [28] [29] (TCP and User Datagram Protocol (UDP)). At the base, the IP protocol provides a data-orientated connectionless means of routing packets from one network to another. As the Internet has developed a number of changes to the IP protocol have been required, with IPv6 currently

[1] Although the traditional Transmission Control Protocol (TCP)/IP model does not utilise a strict layered separation such as the OSI model, the OSI model is used as a conceptual framework for the remainder of this thesis

being rolled out as a replacement to the existing IPv4 platform. At a router level, the major aspect of the change (aside from the new IP header) is the massive expansion in the number of routable addresses provided by the IP specification. Whereas IPv4 utilises a 32-bit address scheme, the newer IPv6 protocol uses 128-bit source and destination addresses. Because the IP protocol does not guarantee delivery and cannot be used to multiplex data from multiple network applications it is common for a higher level transport protocol to be encapsulated within an IP packet. For applications requiring a session-based connection, the TCP protocol can guarantee reliable transmission, creates unique application-based sessions and provides mechanisms for congestion avoidance. For systems requiring multiplexing but without the overhead associated with a state-maintained connection like TCP, the UDP protocol can be used to encapsulate data from the application layer within an IP packet. In addition to these three protocols a number of other protocols are used within the TCP/IP stack, providing functionality such as routing table updates and secure communication.

At a router level, the dominance of IP, TCP and UDP present an advantage by limiting the complexity of the network traffic. Applications such as session-based firewalling, requiring 5-tuple classification (identification using Source Address, Destination Address, Source Port, Destination Port and Protocol), can be efficiently implemented using the knowledge that TCP-based data represents over 80% of the traffic [28] found on the Internet. At a purely engineering level, protocol improvements such as IPv6 present a challenge in two ways. Firstly, routers must be able to support both protocol versions while legacy systems are replaced. Secondly, packet processing techniques such as packet forwarding and classification are optimised for 32-bit address and 32-bit architectures and may not be directly compatible as the number of unique addresses is expanded.

2.2.2 Network Technologies

Lacking any specification of the hardware or link layer details of a network, the TCP/IP suite allows arbitrary communication technologies to be incorporated within the existing

framework. In Table 2.1 some of the communication standards currently in operation are summarised. For local networks, communication can be achieved using both a wired and wireless solution. Wired network technologies such as Ethernet 100 Mbps (100BASE-T) and the state of the art Gigabit Ethernet provide a cheap method of transferring data over short distances. For greater flexibility, wireless equivalents such as IEEE 802.11x can be implemented. For bulk data transfer, optical technologies such as SONET/SDH [30][31] allow network providers to transfer large quantities of data over optical fiber.

Table 2.1: Communication Technologies

Technology	Standard	Bit Rate (Mbps)
Wireless	IEEE 802.11b	11
	IEEE 802.11g	54
	IEEE 802.11n	600
Ethernet	10BASE-T	10
	100BASE-T	100
	1000BASE-T	1000
	10000BASE-T	10000
SONET/SDH	OC-1	51.84
	OC-3	155.52
	OC-12	622.32
	OC-48	2,488
	OC-192	9,953
	OC-768	39,813

In Figure 2.1, we use the network layering methodology outlined in [32] to describe a sample WAN. At the access layer, residential and corporate networks use technologies such as Ethernet, Wireless LAN and PPPoE to connect users. To connect various access networks, the edge or distribution layer allows fast connection between regional networks operating inside the WAN. From a router perspective, access layer data is multiplexed before being transferred over high speed Gigabit Ethernet and OC-3/OC-12 data links. Finally, connection between distribution networks is provided via a core switching network. In this example, the core switching network might represent the network used to connect large metropolitan cities. As in the distribution layer, data is again multiplexed

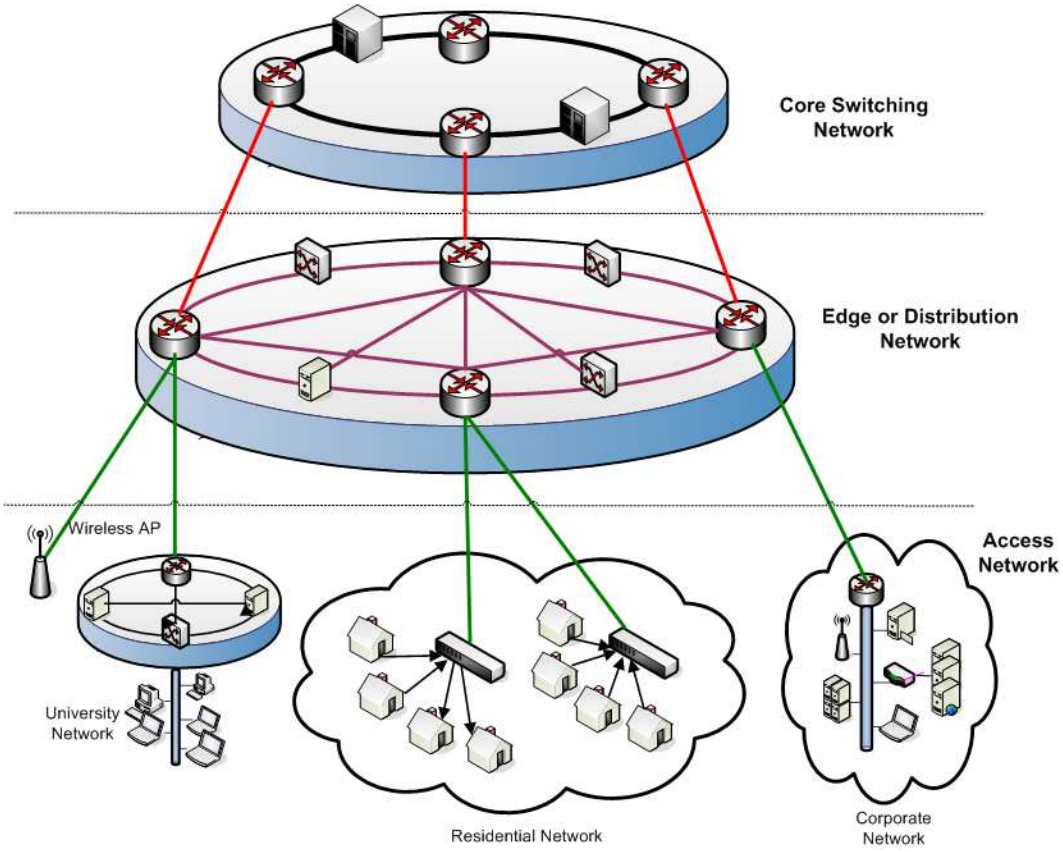


Figure 2.1: Network Layer Topology [32]

before being transferred over high speed links such as 10 Gigabit Ethernet or OC-48/OC-192 connections.

Although a network is divided into three layers in the example outlined above, the separation of layers is not definite and can be altered to accommodate even higher level connections. For example, the TAT-14 OC-768 [33] optical link between the United States of America and Europe provides a means of connecting two large WANs.

Again, from a router design perspective this layer model allows routers to be designed which target specific network topologies. The core network provides maximal switching speed with minimal packet processing. Services provided within the core network are primarily data plane functions such as forwarding, while for routers located within the distribution network, the services provided can be argued to be both data-plane functions such as packet switching, as well as control plane tasks such as congestion avoidance, firewall security and load balancing (e.g. domain name resolution). Finally, routers located within the access network provide services such as web switching, metering, detection

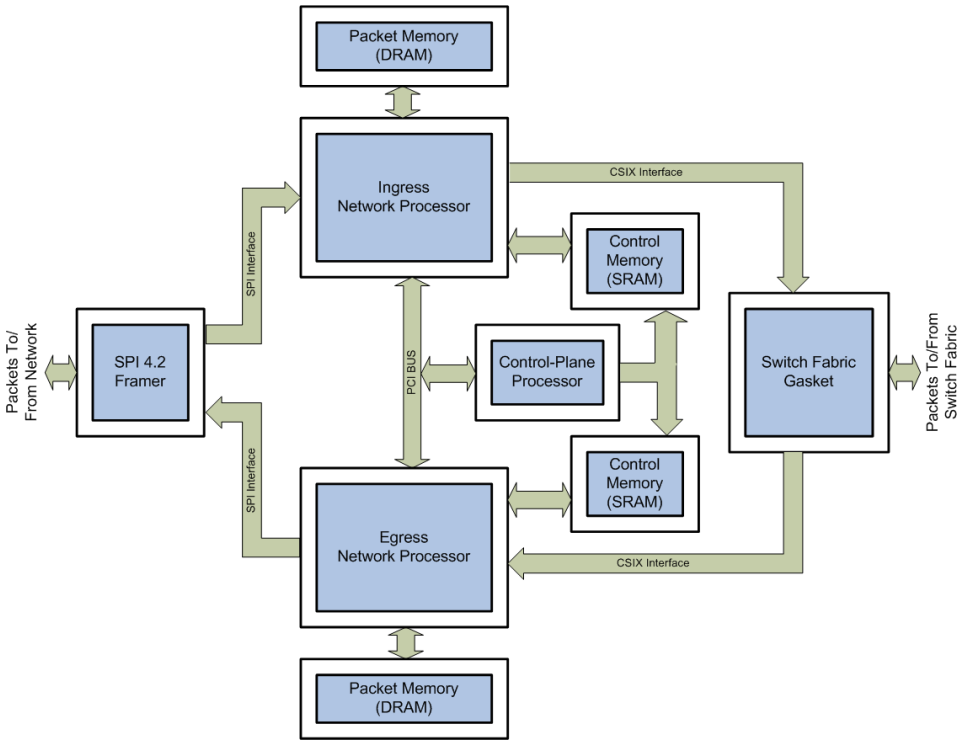


Figure 2.2: Router Line Card [34]

and prevention of network intrusion. The difficulty with such a layered model is that the topology of the Internet is not static. Services such as QoS require routers to be able to differentiate between certain types of traffic at both the access and distribution layer.

2.2.3 Router Architecture

To connect various nodes on a network, packet forwarding or routing devices are needed to maintain paths between one node and another. At its most basic level, a routing node must be able to receive incoming packets, determine the next hop which must be taken and transmit the packet to the corresponding interface. For a programmable NP-based router, a generic architecture might follow the system block diagram shown in Figure 2.2.

Packets arrive via a physical interface, in this case an SPI 4.2 optical interface [35]. Incoming packets are first buffered in the ingress packet memory. The ingress NP will perform ingress tasks such as verification, next hop calculation and packet classification. Once ingress packet processing is complete, the packet can either be transferred to an egress line card on another line card via the switch fabric or it can be transferred to the resident egress network processor. Packets buffered in the egress packet memory are queued

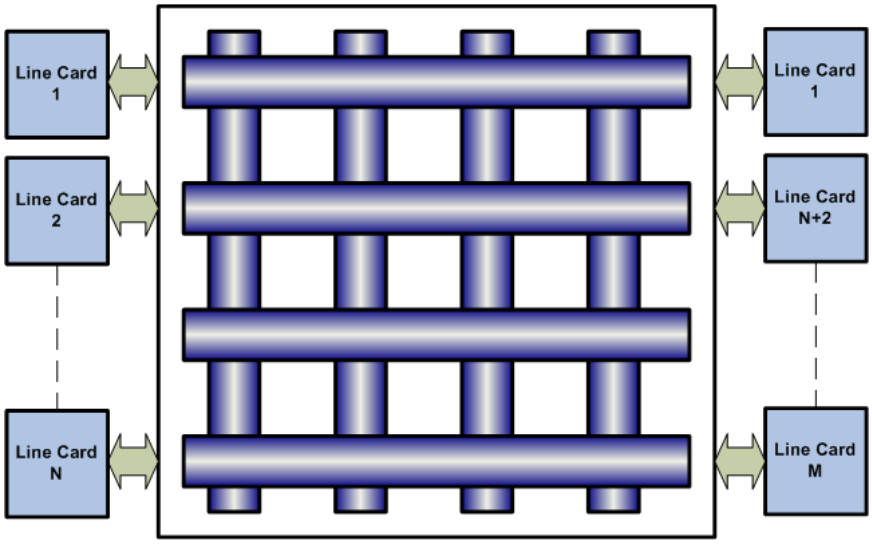


Figure 2.3: Router Architecture

before being processed and transmitted via the physical interface back onto the network. Typical egress processing includes functions such as metering, congestion avoidance, QoS and statistical analysis. An example of such ingress and egress processing might be for the router at the edge of a corporate network to filter unwanted attachments on incoming emails while also removing specific outbound HTTP requests to certain websites.

With a large volume of packets arriving, buffering is commonly implemented with cheap Dynamic Random Access Memory (DRAM) based technologies. To improve performance, faster Static Random Access Memory (SRAM) technology is used to store control information such as routing tables and classification rulesets. For control packets, routing updates and other line card maintenance routines, a control (or Host) plane processor might be provided on the line card. Connection between the control plane processor and the NP(s) is achieved with some form of external communication bus (e.g. Peripheral Component Interconnect (PCI)) or by locating the control plane processor on-chip. The line cards are then connected via the switching fabric, similar to the block diagram outlined in Figure 2.3. The advantage of such a configuration is the scalability which can be achieved with larger switching subsystems and additional line cards.

The system outlined in Figure 2.2 represents just one possible NP line card configuration. Certain architectures implement the control plane within the NP die, reducing the line-card cost, while other NP designs implement an external bus connection, allowing

the customer to decide the type of control plane architecture employed. Similarly, an NP architecture can be designed to utilise a single type of memory while another design may utilise a highly segmented memory architecture.

2.3 Network Processors

While the line card outlined in the previous section represents the most common method of building a modern router, the underlying NP architecture on the line card varies greatly from one device to another. In both academic and commercial research no single NP architecture has emerged as the optimum configuration, but a number of traits have emerged which can be argued to represent the most common features found within an NP architecture. To highlight these features two commercial NP architectures are briefly examined in this section.

2.3.1 Intel IXP-28XX Network Processor

One of the most flexible NP architectures, the IXP-28xx [34] line of network processors incorporate a number of various architectural and technological improvements over the legacy IXP-12xx NP. Utilising 16 ultra high performance PEs (labelled Micro-Engines by Intel), each running at 1.4 GHz, the IXP-28xx platform includes a 700MHz on-chip Xscale processor. With memory and I/O access speed failing to keep pace with microprocessor development, a number of mechanisms have been employed to either mitigate or hide this access latency. Firstly, the memory hierarchy employs multiple channels for both control and packet memory. Secondly, each PE is hardware multi-threaded, supporting 8 threads. Connection to the external device can be achieved via three interfaces. Physical devices are connected via the Serial Peripheral Interface (SPI) 4.2 [35], while the Common Switch Interface (CSIX) connects the NP to the switching fabric. Inter-process communication is provided via the PCI bus. While communication between arbitrary MEs remains, the IXP-28xx architecture favours either a pipeline or parallel-pipeline architecture, with low latency communication possible between an ME and its ‘next neighbour’.

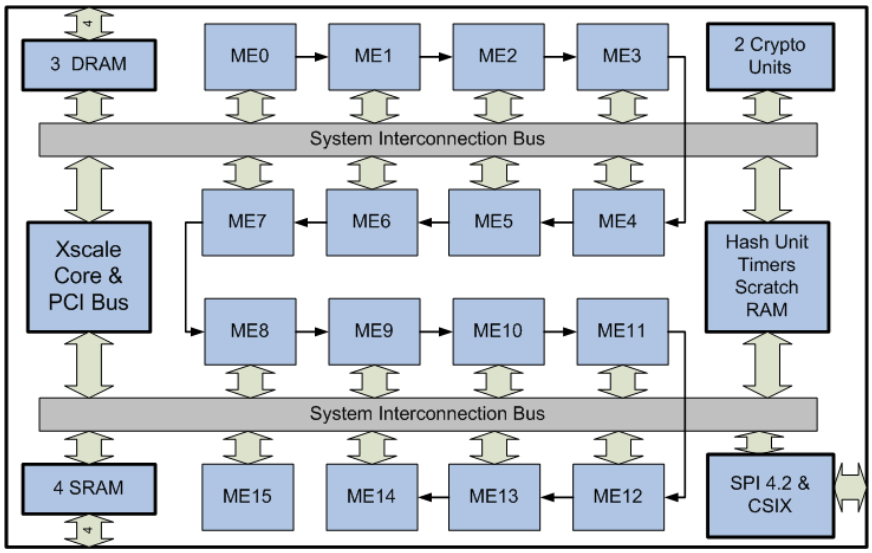


Figure 2.4: The Intel IXP 2805 (From [34])

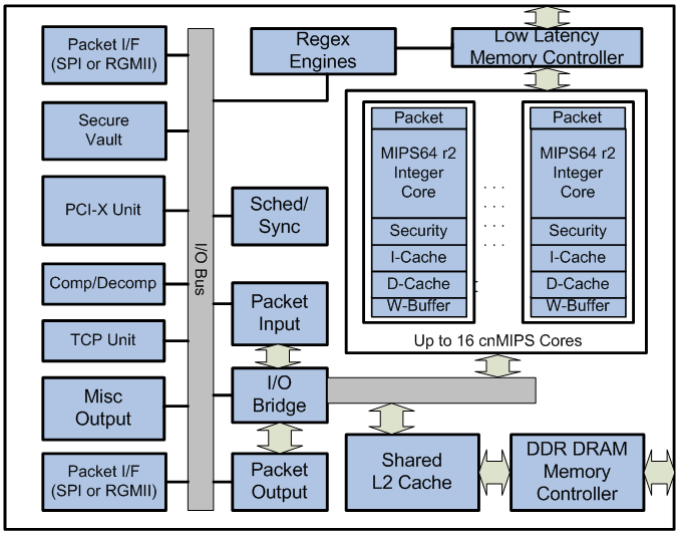


Figure 2.5: Cavium Octeon Cn58XX (From [36])

For network applications requiring packet security, the IXP-28xx family of devices can include on-board cryptographic blocks, allowing computational intensive encryption and authentication to be offloaded to dedicated hardware.

2.3.2 Cavium OCTEON Cn58XX

Like the IXP-28xx platform, the Cavium OCTEON Cn58xx [36] NP represents a full System on Chip (SoC) architecture. Supporting up to 10Gbps in full duplex mode, the Cn58xx architecture includes a number of hardware accelerators aimed at mitigating some of the performance loss associated with computational intensive tasks within a network

router. The NP can be interfaced to either a Reduced Gigabit Media Independent Interface (RGMII) or SPI-4.2 physical device, with packets and control information stored in either low latency second generation Reduced Latency DRAM (RLDRAM2) or higher latency Double Data Rate DRAM (DDR2-DRAM). Data plane processing is provided by between 4 and 16 cnMIPS microprocessors.

Derived from the MIPS64 [22] RISC architecture, the Cavium architecture is unique in that the data plane processing elements include a multi-level cache hierarchy. Along with the cache structure, each cnMIPS core includes hardware accelerators for functions such as encryption and authentication. Shared hardware accelerators include a TCP Unit to offload TCP maintenance functions and a data compression/decompression unit conforming to RFC1950/51 [37]. Finally, the 32 Regular Expression engines allow parallel deep packet inspection.

2.3.3 State of the Art NP Architectures

Summarising the two architectures outlined in previous section, it can be seen that the major trade-off within NP design is how to maximise performance while maintaining flexibility. Providing a purely PE-based dataplane would maximise flexibility but certain tasks such as encryption are difficult to implement in software at the required performance levels. On the other hand, an architecture in which the PE only provides basic functionality in software with a large amount of hardware accelerated blocks would limit the ability to update the services and algorithms implemented on the router. Both the Intel and Cavium architectures utilise a RISC-based PE structure with hardware blocks augmenting the programmable data plane. Examining other commercial architectures, the EZchip NP-1 [38] and Xelerated X11q [39] can be argued to represent two architectures in which design flexibility is reduced in order to maximise performance. Both utilise a strictly pipelined layout (in the case of EZchip the architecture is parallel-pipeline), with the PEs heavily optimised for NP applications. Each stage of the Xelerated architecture can execute only 4 instructions. Despite the large variation across commercial architectures, a number of architectural techniques have evolved to become common features within network

processor design.

2.3.3.1 Multicore Processing

At both a packet and flow level, parallelism provides a method of scaling performance to meet higher bandwidth demands. Although the number of PEs in most NPs has been around the order of 4-16 ([34],[36],[40]), a number of architectures stand out as employing a substantially higher degree of parallelism. Firstly, the Netronome NFP-3200 [41] and Cisco QuantumFlow [42] utilise 40 processing engines. The superscalar design employed by EZchip in the NP-2 and NP-3 [43] NP also exploits large scale parallelism (the number of PEs is not available in the public domain). The Xelerated X11 NP provides neither flow nor packet level parallelism and instead uses 800 Packet Instruction Set Computers (PISC) engines arranged in a linear pipeline [39]. Furthermore, the Silicon Packet Processor (SPP) developed by Cisco includes 188 separate RISC engines per chip [44].

2.3.3.2 NP Interconnection

Despite NPs employing a multiprocessor architecture, there has been little research into bus topologies for NP systems. Weng and Wolf proposed a mechanism for distributing tasks across a parallel system but the underlying bus parameters were not examined [45], while Karim et. al. is, to the author's knowledge, the only published work to examine bus topologies in NP systems [46]. For general purpose systems, communication systems between processors, memory and external devices has undergone more extensive research, with proposed and available bus systems optimised for low-cost [47], high performance when streaming data [48] or universality [49].

2.3.3.3 Integrated Networking Interface

To reduce the latency associated with packet ingress and egress operations, networking interfaces are integrated on-chip instead of being bridged to the network processor. Typical implementations include a means of connecting the network processor to a physical layer device such as a Gigabit Optical, Gigabit Media Independent Interface (GMII) or Packet

Over SONET (POS) SPI-4.2 interface [50]. Along with this physical device interface, some cases will include the Common Switch Interface (CSIX) to provide interconnection of the NP and a router switching fabric.

2.3.3.4 Multithread Processing

Providing a cheap method of minimising the cost associated with long latency operations, multi-threading lends itself to network processing in a number of ways. Firstly, the parallelism outlined in Section 2.3.3.1 can be extended to include multi-threads. Secondly, applications suited to pipeline partitioning can be implemented as either a pipeline across multiple PEs or as a conceptual pipeline located on a single PE. Finally, multiple threads or contexts are reasonably cheap to implement. Requiring little interaction between the programmer and application, a context swap can be achieved by a single register swap, with only a small amount of control hardware needed to schedule each thread.

2.3.3.5 Control Plane (Host) Processing

In addition to the data-plane processing, routers also require a means of providing control plane functionality. At a fundamental level, these control plane tasks include functions such as routing table updates, classification ruleset updates and adjustments to the scheduling functions, as well as processing non-standard packets (e.g. control information from other routers). More complex functions require implementing an operating system on the host Central Processing Unit (CPU) as a means of providing system control across the line card. While the Intel, Netronome and HIFN architectures implement an on-chip control plane processor, the remaining NPs typically provide either a specific port or a generic PCI interface between the NP(s) and a host CPU.

2.3.3.6 Integrated Memory Controller

For general purpose processing that is not sensitive to access latency, the memory subsystem is optimised for bandwidth rather than latency. Due to the low latency requirements of packet processing, NPs must perform fast memory operations in order to match the

minimum packet inter-arrival time [51]. Therefore, most NPs have integrated memory controllers in order to achieve lower latency. Furthermore, the low spatial locality of network applications also suggests that the optimisation of bandwidth for memory subsystem is not as effective in NP as in GPP architectures [52].

2.3.3.7 Hardware Accelerator

Offloading special applications that are relatively stable and suitable for hardware implementation has been adopted as an important method to achieve high performance. Hardware accelerators usually function as coprocessors and have the potential of being executed concurrently with other parts of the program. They can be implemented either private to or shared by the processing cores, or implemented as external devices interacting with the NP. Hardware acceleration is covered in more detail in section 2.5.2.

2.4 Network Processor Based Applications

Following on from an overview of NP architectures, an overview of typical NP applications is presented in order to provide a context for NP development. As was mentioned in Chapter 1, the field of NP-based applications is notable for its growth from simple packet switching to the complex packet processing functions implemented today. In general, the trend has been for applications to be pushed down the network stack, with the router providing functions between layer 3 (packet forwarding) and layer 7 (web switching, Deep Packet Inspection (DPI)). Although it is more common to provide services such as firewalling, classification and inspection at an access level, research within this topic has largely been targeted at providing such functionality at routers located within both the edge and core networks. For example, [53] and [54] are two works which have examined implementing five-tuple packet classification and deep packet inspection at ultra-high speed OC-192 and OC-768 line rates respectively. In this section we present a detailed analysis of how the application growth outlined in Section 1.2.3 affects NP research. In general, NP applications can be divided into two basic classes. The first class comprises

those applications which use only the packet header during processing (*Header Processing Applications*). These applications include packet forwarding, packet classification and packet metering. The second class of NP applications is comprised of those applications which utilise both the header(s) of a packet and the packet payload (*Payload Processing Applications*).

2.4.1 Packet Forwarding

Packet forwarding remains the fundamental task of any router, but the complexity associated with packet forwarding has been increased by two trends. Firstly, global network growth will quickly exhaust the 32-bit address space provided by the IPv4 protocol, with IPv6 expanding IP address space to 128-bits. At a router level, the challenge will be how longer addresses can be traversed and stored. Current longest prefix matching algorithms have been optimised to process 32-bit network host addresses and 24-bit prefix masks, utilising 32-bit memory to store each node of the routing table.

In parallel with the development of IPv6, the number of addressable networks deployed has grown at a quadratic rate, with the number of Border Gateway Protocol (BGP) entries in the Internet routing table expanding from 50,000 entries in 1998 [55] to almost 300,000 entries in the current table [56].

2.4.2 Quality of Service

Fundamental to providing high priority services such as VOIP is the ability to differentiate various packet flows within IP traffic. A flow is defined as a sequence of packets transferred between two computers. At a coarse level, a flow may be identified using the source and destination addresses of these two nodes. With applications such as Network Address Translation (NAT) [57] mapping multiple terminals to the same outgoing IP address, it can be seen that such two tuple flow classification would be ineffective at correctly identifying an application flow. A more finely grained solution exploits the dominance of the TCP and UDP protocols to identify flows using both the IP addresses and port addresses. Algorithms such as [58] [59] [60] [61] [62] allow five-tuple classifi-

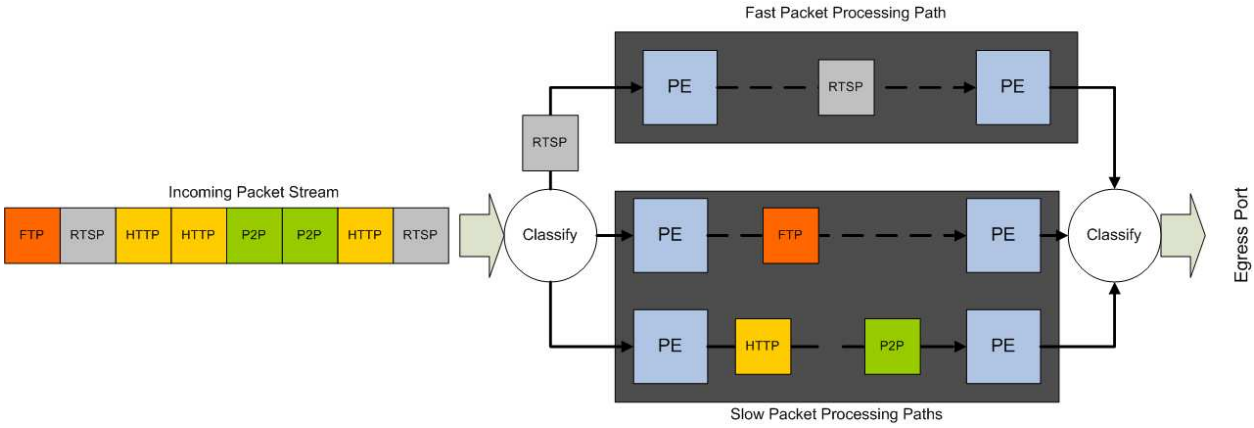


Figure 2.6: Classifying Router

cation to be implemented at router level. Five-tuple classification allows network service providers to implement services such as QoS, session based firewalls or metering at wire speed. Figure 2.6 describes a typical NP-based QoS system in which packets are classified before being directed to either a fast or slow processing path. While latency dependent Real Time Protocol (RTP) applications can be directed to the fast path, applications with little or no need for low latency, such as Peer-To-Peer, can be directed to the slower path. With future growth in mind, the question is whether such five-tuple solutions are scalable.

A more significant challenge to both QoS and packet classification is the blurred nature of how new applications are implemented on an IP switched network. The popular YouTube video hosting website demonstrates this point. In the case of YouTube, videos are 'streamed' to the user using HTTP as opposed to a streaming specific protocol such as Real Time Streaming Protocol (RTSP) [14]. As such, a classification rule must be able to differentiate between traditional HTTP traffic on port 80 and this streamed data. The second difficulty with 5-tuple classification can be seen in an application such as Skype [63] . Although a closed source protocol, the Skype protocol uses an RTP-based application in which audio and video information are encapsulated using either TCP or UDP. At runtime it is possible to configure Skype to utilise any port when communicating, making even 5-tuple classification difficult. Given the availability of an option of securing packets between users, it was shown in [64] that identifying skype applications is a non-trivial task, requiring deeper packet inspection.

2.4.3 Security

In general, network security can be divided into two categories. The first aspect of network security involves mechanisms for securing data transported over an open network. As well as securing communication channels, network security also requires the detection and prevention of attacks or infiltration by persons located outside the specific network.

2.4.3.1 Packet Security

The movement of functions such as communication and commerce to the Internet has facilitated the need for secure channels between various networks. The channel can be either between networks in which a VPN might be available or from one user to a secure server in which personal data must be protected during transmission. Using the Internet Protocol Security (IPsec) security suite [65] along with transport layer protocols such as Secure Socket Layer (SSL) and Transport Layer Security (TSL) [66] it is possible to provide secure data transmission, authentication and key exchange across an unsecured network. For networks such as those employed by multi-national corporations the security trade-off can be demonstrated when examining communication between two geographically separate networks. Security can be either be provided by each node within both networks, (i.e. all nodes have access to the encryption keys) or both networks can be configured with routers automatically securing data transmissions between both networks in a fashion transparent to end users. With a high volume of traffic being sent between these networks, the VPN performance provided by the routers must be significant. The IPsec protocol commonly used to provide packet security is algorithm-agnostic; listing many possible algorithms and allowing the user to select which security algorithms are used. Common algorithms used for encryption and authentication include the Advanced Encryption Standard (AES) [67] symmetric encryption algorithm and the Secure Hash Algorithm (SHA) [68] hashing function.

2.4.3.2 Network Security

Network security encompasses techniques aimed at protecting a network or computer from infiltration. The most common example of this is the network firewall. While a personal computer can be protected via a software firewall implemented on the computer, protecting an entire network necessitates the implementation of a high performance firewall on the edge router. Older firewalls attempted to filter unwanted traffic by applying broadly based rules on the incoming packets. More intelligent firewalls attempted to retain state-based flow information when examining network data by using a 5-tuple classification algorithm, such as those outlined in Section 2.4.2, to identify packet flows before applying the matching rule. More modern firewalls have implemented deep packet inspection techniques as a more finely grained means of matching packets and rulesets.

While a standard 5-tuple packet classification scheme can be used to identify malicious flows, it does have some serious limitations when used to construct a layer 4 firewall. For example, a firewall may implement a website blacklist which blocks outgoing TCP connections to a known IP address, but techniques such as DNS misdirection, or IP spoofing, can make a 5-tuple based ruleset cumbersome to implement, easy to bypass and difficult to maintain. A more intelligent technique is to implement a payload based detection system. Commonly referred to as DPI (or Network Intrusion Detection System (NIDS)), the goal of such a system is to allow packets to be classified using both session information and payload data. An example might include a rule which attempts to filter incoming traffic containing a new virus. As in five-tuple schemes, packets arriving at the network interface are examined against a predefined ruleset, with the payload examined for a byte-string similar to the virus definition. Another use may be to detect unauthorised access to the network, for example examining the commands issued during a telnet session. This is a popular topic within current NP research and many methods and solutions have been proposed, [69], [54], [70], [71], with certain algorithms tailored to hardware implementations while others utilise a structure more suitable to software.

2.4.4 Payload Based Applications

A number of other payload-based applications have been developed which might be implemented at a packet level. Firstly, web or content switching provides a means of load-balancing a web site which is hosted over a number of physical servers. With busy websites processing millions of requests every second, the workload associated with these connections is distributed over *n* hosts. The task of a content or web switch is to extract important information from incoming data (HTTP Requests) as a means of balancing the demands placed on each server. For popular websites, load-balancing can be seen as a non-critical design aspect which aims to improve server utilisation while limiting access latency, whereas a payment processing site would be highly sensitive to delayed transactions during busy periods.

In addition to content aware switching is the possibility of implementing media transcoding at a network level, although only a little research has been undertaken in this area [72]. For example, with an increasing amount of video information stored on social media sites, a typical transcoding scheme might entail storing the videos in a high definition format in the database. Once requested for streaming, the video is transcoded by the outgoing network router, with the selection of the encoding bitrate dictated by network conditions.

2.5 Scalability of Network Processors

Given the growth in both NP application complexity and network demands, a fundamental question is whether the NP-based solution can be scaled with future needs in mind. In this section an outline of the current state of the art within NP research with respect to scaling NP architectures is given. Some of this research can be argued to be at a *macro* level, providing additional resources (e.g. more PEs, hardware acceleration) while other methods typically focus on improving the performance on one module of the NP structure.

2.5.1 PE Parallelism

Examining any commercial architecture, the most obvious performance increment which could be investigated is to increase the number of PEs employed in the NP architecture. With packet data well suited to parallelism, increasing the number of PEs has been well exploited in both commercial and academic research. Within the commercial domain this Packet Level Parallelism (PLP) can be seen in all NP architectures except Xelerated X10 and X11 NPs which implement a massively pipelined design. While the challenges regarding load-balancing and packet re-ordering are solved via on-board hardware accelerators in the Cavium NP, the IXP based architectures (Intel, Netronome) leave scheduling issues to be resolved in software via a number of ingress PEs.

In general, the amount of PLP available to a network processor designer is bounded by two factors. At a hardware level, the provision of additional PEs requires changes to the bus and memory structures. Additionally, the need to retain correct packet order requires a router to retain a reordering mechanism. Research within the topic of PLP has tended to give mixed results. In [73] it was found that multi-processor systems provide additional performance, although the simulation was limited to 8 separate PEs. In [74], Gries et al. demonstrated that a parallel architecture is both easier to programme and to scale, with IP forwarding applications suited to parallel architectures. While [75] found that PE utilisation falls off significantly as the number of PEs is increased, Thread Level Parallelism (TLP) can provide more scope for performance improvements. Research by Shi et al. [76] found that organising the PEs in a clustered format provides a mechanism for increasing parallelism but requires additional logic to handle both the load-balancing and packet reordering. The challenges involved in packet re-ordering at ultra high bitrates has been extensively studied, with numerous papers highlighting the challenge as well as proposing both hardware and software solutions (see [77], [78], [79] and [80]).

2.5.2 Hardware Acceleration

With dedicated logic able to maintain higher throughput when compared to a programmable solution, another method of scaling performance is by offloading complex and computa-

Table 2.2: Hardware Implementations of the AES Algorithm

Architecture	$N_{pipeline}$	A_{slices}	f_{max}(MHz)	T_{put}(Gbps)
SBOX	1	1168	125.251	16.032
SBOX	2	1233	156.568	20.04
SBOX	3	1297	183.083	23.434
CFA	1	1306	100.766	12.899
CFA	2	1290	107.4	13.747
CFA	3	1148	136.036	17.412
CFA	4	1086	147.449	18.873
CFA	5	1121	159.363	20.398

tional intensive tasks to dedicated hardware. Traditionally, the offloading of network tasks to hardware was limited to tasks requiring extensive processing in software, e.g. encryption and authentication. The hardware block can either be coupled to each PE to create low contention co-processors or can be shared across all PEs. While co-processor based architectures provide better performance, the additional silicon cost of per-PE solutions is significant. For example, Table 2.2 summarises the performance of Field Programmable Gate Array (FPGA)-based implementations of the AES algorithm [81].

When implemented in FPGA logic, it is possible to map the underlying non-linear lookup functions to either local memory within each slice or to the global memory distributed throughout the FPGA. Each architecture is labelled to reflect the underlying architecture and round latency per block. The SBOX architecture maps 256x8-bit tables to local FPGA SRAM, while a description of the Composite Field Arithmetic (CFA) AES architecture can be found in [82]. As can be seen from the results, only a small amount of pipelining is required to obtain a throughput in excess of 20 Gbps. In general, the SBOX architecture is well suited to modern FPGA since the on-board memory can be used to store the substitution tables used during encryption. Requiring an average cost of $\sim$1200 Xilinx CLB slices to implement, the cost of employing dedicated on-chip hardware can be expensive, especially when considered against the $\sim$2500 CLB slice Leon2 5-stage microprocessor [83]. On an ASIC platform, the work in [84] outlined an architecture which trades throughput for size, requiring approximately 58.5K transistors to implement, it can

Table 2.3: Survey of Commercial Hardware Acceleration Solutions

Architecture	**Encrypt & Authenticate**	**Meter & Queue**	**Forward & Classify**	**Misc**
IXP-2805	3DES,AES SHA-1			CRC[1] HASH
Cavium CN58XX	3DES,AES SHA/MD5[1]	Sched Sync		Regex TCP/Comp
AMCC np3710		WRED, WRR Shaping,Priority	Policy Engine	Hash Stats
HIFN 5NP4G		TrTCM	Classify Trie	VLAN CRC[1]
EZchip NP-2		LBM,WFQ Priority,WRED	Search	
Bay Montego		LBM	TCAM WRED	SAR
Xelerated X11d		Priority	TCAM, Search Look-Aside	
Wintegra WinPath2	3DES,AES SHA,MD5	WRR Priority		CRC NIDS
LSI APP650		Cell,WRR WFQ, Priority	Policy Engine	SAR Stats

provide peak performance of 2 Gbps. On the other hand, a full system on chip version of the Leon2 core requires only 35K transistors to implement.

Table 2.3 presents a survey of the current state of the art in hardware acceleration employed by commercial NP manufacturers. As can be seen in the table, metering, congestion and queueing algorithms represent the most common functions to be implemented in hardware. With the exception of Intel, all other manufacturers implement some configuration which may involve queue maintenance, metering or congestion avoidance.

Most architectures implement a hardware block which can be configured to process either variable length traffic or cell basic protocols such as Asynchronous Transfer Mode (ATM). Following on from this, datagrams can be queued using a load balancing algorithm such as Weighted Round Robin (WRR) or by defining certain queues for higher priority traffic. While packet forwarding and classification schemes seem popular, it should be noted that a number of accelerators amount to little more than a specific memory inter-

face which can be connected to a high speed search device such as a Ternary Content Addressable Memory (TCAM). Uniquely among commercial solutions, Cavium implement the encryption blocks as primitive functions on a per processor basis. With no access contention, per processor designs allow high performance provided the packet data is stored close to the PEs also. Only two architectures (Wintegra, Cavium) currently include a deep packet matching scheme. Finally, marking and congestion avoidance algorithms such as Three Color Marker (TCM) or Random Early Detection (RED) can be utilised to ensure minimum performance under heavy network loads.

2.5.3 Technological Evolution

Alongside performance increases in parallelism, additional processing performance can be achieved by harnessing the performance increases seen in each new Complementary Metal Oxide Silicon (CMOS) generation. According to Intel, CMOS technology increments between one silicon generation and the next allow transistor performance to be increased by approximately 1.5 [85]. With future demands in mind, two significant challenges are apparent. Firstly, the ability to scale CMOS technology is becoming increasingly difficult with future CMOS generations requiring extensive structural and technological changes [86]. Along with these manufacturing challenges, the increase in clock frequency associated with technology changes increase both static and dynamic power consumption. With power consumption becoming ever more important in digital electronics, performance gains must be examined with respect to the additional energy-based running costs associated with the device.

2.5.4 PE Specific Techniques

In addition to these macro level techniques there are a number of PE specific techniques which could be either employed or optimised in order to improve NP performance. Clearly one method of improving PE would be to utilise the technological evolution briefly outlined in the previous section, however there are a number of additional micro-architectural techniques which can be employed at a PE design level.

2.5.4.1 Instruction Level Parallelism (ILP)

Although computer programs execute in sequential order there remains some independence between differing programs (or areas of the same program). Instruction Level Parallelism (ILP) attempts to exploit this independence by implementing a superscalar type design within the CPU (a more in-depth background into superscalar CPU design can be found in [87] and [88]). Superscalar designs typically implement a mechanism for dispatching multiple instructions at once provided there is no data dependency between the instructions. There are, however, a number of limits to the degree of parallelism which can be extracted. Firstly, clearly not all instructions will be independent, with data and resource dependencies common to modern programs. Secondly, while out-of-order completion allows greater parallelism, issues regarding re-order buffers, register renaming, virtual registers, etc. are introduced to the CPU design [89]. Finally, a coarse superscalar design incurs a significant penalty during branch mispredictions. Within research targeting network processor performance, only a small amount of work has been undertaken in the area of ILP for NP architectures. Memic et. al. found that network applications exhibit a lower degree of ILP when compared to media based applications [90], while other research found that ILP was better suited to control plane functions [91]. Within general purpose processing, architectures such as the Intel Pentium 4 implement a dual speed Arithmetic Logic Unit (ALU) design [92], in which instructions are dispatched to either a fast ALU when only simple bitwise or arithmetic operations are required, or to a slower ALU designed for operations such as multiplication or bit positioning and manipulation. It remains unclear if such techniques are justified on NP platforms.

2.5.4.2 Thread Level Parallelism (TLP)

Within general purpose processing, thread level parallelism is typically implemented as a microprocessor supporting multiple contexts or threads executing on the same processor. In [93] it was found that multi-threading allows greater microprocessor utilisation provided enough bus and memory bandwidth is available. In a multi-processor environment this problem is compounded by the fact that an n-thread by m-processor system

can easily saturate shared devices. Within network processing, thread level parallelism can be implemented as either inter-packet level parallelism or intra-packet level parallelism. Inter packet TLP operates by running the same application on all threads, while intra-packet level parallelism involves exploiting operational independence within certain network applications. An example of this independence can be seen in the address verification routines required for IP forwarding, with separate threads created for verifying the source and destination addresses.

Inter-packet TLP can be viewed as increasing the number of effective PEs by providing a framework for increasing PE utilisation. With m threads operating on n PEs in parallel, the primary difficulty with inter-packet TLP is the higher device contention triggered by each additional thread. On the other hand, the more finely grained intra-packet TLP provides the programmer with a mechanism to balance contention issues and utilisation at the cost of additional task partitioning. Within academic research, an analytical model developed in [94] found that the limit on TLP is bound by certain sections of the underlying program, while [95] found that the limitation on TLP can only be lifted through higher clock frequencies. In [96], a heuristic methodology for mapping network applications to multi-processor and multi-threaded NPs was presented, with substantial performance increases possible.

Within the commercial domain, various architectures demonstrate the difficulty in using TLP as a means of hiding latency. While AMCC nPcore supports 24 threads, the IXP family of network processors utilise 8 threads. On the other hand, the OCTEON architecture employs a cache structure between the PEs and main memory, with no multi-threading employed.

2.5.4.3 Caching

Another architectural aspect by which NP performance can be improved while retaining programming flexibility is through a cache-like structure between the PE array and external system memory. Within general purpose processing caching has been increasingly important as memory access speed has failed to keep up with CPU operating frequency.

With operating systems already multi-threaded, more intelligent cache hierarchies provide one mechanism for reducing memory access latency. Only the Cavium architecture employs a cache hierarchy on a per PE basis. Each cnMIPS engine employs a 32-KB instruction cache, 16-KB data cache and a 2MB shared level 2 cache. Uniquely among commercial NPs, Cavium does not couple the program memory close to the PEs, necessitating a cache structure between the PE and memory in order to avoid long delays during instruction fetch operations. Academic research within the topic of caching for NPs has tended to give mixed results. While Lekkas [97], [98], Comer [91] and Zhen [99] either find or state that the poor spatial and temporal locality of network traffic limits the performance gain of a cache hierarchy, the work by Wolf and Franklin in [100] and Memik et. al. [90] found that cache sizes up to 32-KB do provide adequate cache hit rates, although only for certain network applications. On the other hand, the work in [51] (expanded in [101]) found that cache can provide improvements in NP performance when coupled with multi-threading.

Although it is difficult to ascertain the reason for wide variance across research, a number of variables are possible. Firstly, locality within a network data stream will be determined by the nature of the IP traffic traversing the router. The presence of either large or small amounts of active flows, as well as the duration of the flows, will affect data locality within the cache. Following on from this, the use of various microprocessor architectures across studies limits the ability to make comparisons, with differing architectures providing different memory functions (e.g. multi-word operations) and different register file sizes.

2.5.5 Summary of NP Scalability

This section outlined the various methods that are available by which future NP demands could be met without having to return to an ASIC solution. Provided compiler and software development can be improved to reflect parallel work flows, it is expected that increasing parallelism will continue to provide one method of scaling NP performance, within the bounds of Amdahl's law [102]. At a PE level it is possible that existing CPU

techniques such as ILP or caching could be modified to better perform on NP platforms. Although not mentioned above, another method by which PE performance could be increased is by implementing a deeper processor pipeline in order to increase the operating frequency of the PE. With a large proportion of this thesis dealing specifically with this branch penalty the next section presents a detailed background into the topic of micro-processor pipelines, as well as outlining some of the penalties associated with a pipeline microprocessor design.

2.6 Micro-Architectural Considerations of PE Design

In general, NP platforms have employed a RISC-based architecture for each of the PEs. Although there are many advantages and disadvantages of a simpler RISC platform over a more complex Instruction Set Architecture (ISA) for an NP platform, the primary benefit remains the lower silicon cost. The provision of complex instructions typically requires additional logic in silicon, for a parallel system such as an NP these additional transistors scale with the degree of parallelism employed. Although the RISC architectural philosophy originally proposed only simple instructions to be supported in hardware, modern application and performance demands have required functions such as hardware multiplication or multi-threading to be incorporated.

Summarising both commercial and academic research it is possible to define a PE as being a Harvard-based, cache-less, integer-only RISC micro-processor, with design aspects such as multi-threading or memory management varying between designs. Similar to the engineering trade-offs within general purpose processing, an NP designer can choose to either implement a proprietary ISA which requires extensive software development (compilers, assemblers, etc), or take advantage of a new trend within microprocessor design in which the base ISA of an architecture is specified but each customer is allowed to add new instructions to this base ISA. Within the commercial domain, the Cavium and Cisco NPs implement PE architectures derived from MIPS and Tensilica cores respectively, while the Intel IXP, Netronome and AMCC implement custom RISC architectures. In performance terms, the design and implementation of new ISA allows the architecture

to be optimised for NP applications, adding instructions which are expected to be heavily utilised. For example, two instructions important to NP platforms are the population count *POP*, which returns the number of bits set in a data word, and the Count Leading Zero *CLZ* instruction, which is used to determine the bit position of the first 'one' in a data word.

While a new ISA provides some performance increases, the re-use of an existing ISA allows the existing code base and toolset to be exploited. Today, a version of the Linux OS has been ported to the ARM, MIPS and Tensilica architectures, allowing networking libraries and routines to be re-used, while also providing mature development tools such as the GNU Compiler Collection (GCC) [103]. In addition to this ISA architectural choice, a second design consideration is how each instruction within the ISA can be efficiently partitioned in order to take advantage of the performance benefits which can be obtained through pipelining.

2.6.1 Pipelining

Within a combinational digital system the maximum operating frequency of the circuit is defined by the slowest path between any two edge triggered registers. Assuming both registers are triggered on the same clock edge (either rising or falling), the maximum operating frequency can be determined using equation 2.1

$$f_{max} = \frac{1}{t_{co} + t_{wire} + t_{logic} + t_{su}} \tag{2.1}$$

where t_{co} is the propagation delay between a registered input value and the time that this value is carried to the register output, t_{wire} is the wire delay of the critical path, t_{logic} is the logic delay of the critical path (typically the number of NAND gates located in the path) and t_{su} is the amount of time the final value must be stable at the output register in order to be registered correctly. While t_{co} and t_{su} are technology defined limits, the wire delay (t_{wire}) is determined after design by the layout or synthesis tools. As such, the only mechanism for increasing performance is through a reduction in the logic delay. This con-

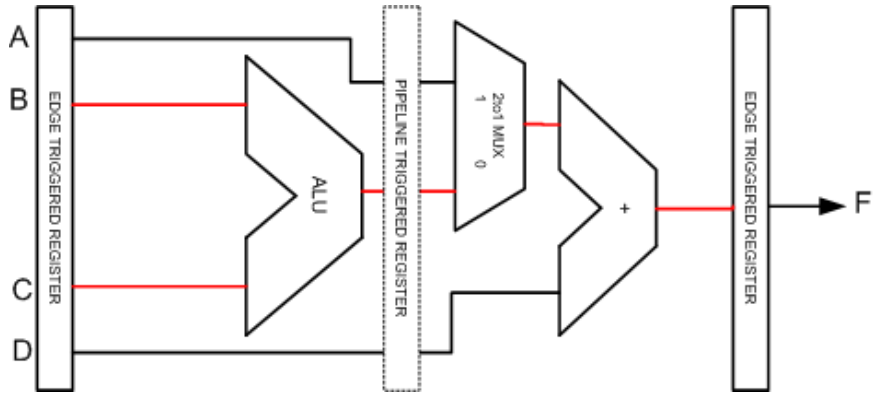


Figure 2.7: Pipelining of a Combinational Circuit

cept is illustrated in Figure 2.7 where both the wire and logic delays are reduced through the addition of a pipeline register. Obviously the goal of any additional pipeline registers within a digital design should be to maximise the obtainable frequency by evenly dividing the critical path either side of the register. A further example illustrating the benefits of pipelining can be seen in the performance of complex algorithms when implemented in hardware, such as encryption. Requiring n cycles to process p data blocks, a non-pipelined device operating at frequency f_{clk} incurs a latency given by equation 2.2.

$$t_{non-pipe} = \frac{n * p}{f_{clk}} \tag{2.2}$$

While implemented as a non-interlocked pipeline (n=*1*) of s stages, the processing delay for the same p blocks is calculated using Equation 2.3.

$$t_{pipe} = \frac{s + (p - 1)}{f_{clk}} \tag{2.3}$$

When implemented at a micro-architectural level, instruction pipelining attempts to divide a single operation, the instruction, into a number of smaller overlapping operations. Figure 2.8 demonstrates the common DLX 5-stage pipeline [52] [2]. In this case the instruction is divided into five separate operations, which can be implemented as an overlapping pipeline. To ensure maximum throughput, the goal of pipelining is to divide the combinational operations so the delay of each stage is minimised. While the original ar-

[2]The remainder of this thesis uses this traditional 5-stage pipeline as a reference design.

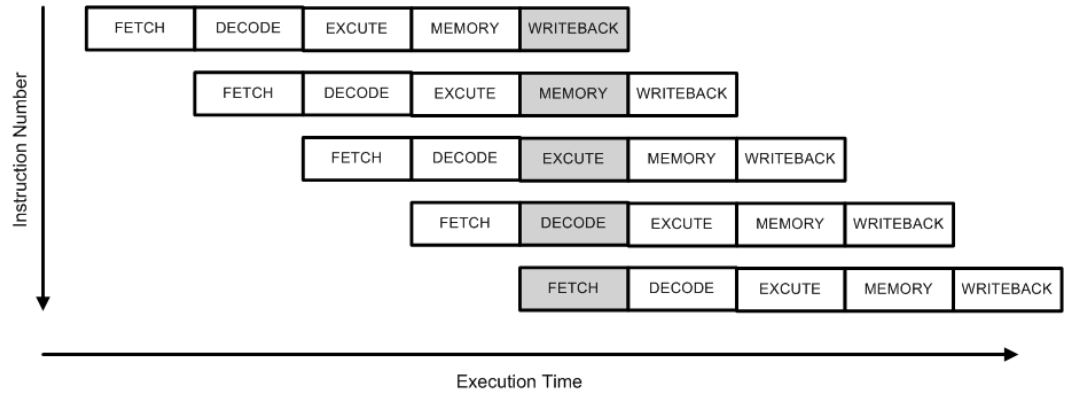


Figure 2.8: 5-Stage Integer Pipeline

chitecture outlined in [52] solved this problem by removing any instructions which could not be efficiently divided into five stages, more modern architectures have accepted the need to provide complex multi-cycle functions such as multiplication, shifting or address generation. Pipelining is one architectural method of improving PE performance, provided a mechanism for controlling power consumption at higher frequencies is included and the performance limitations associated with hazards are solved. The next section presents an overview of these pipeline hazards.

2.6.2 Pipelined Architecture

For a load/store architecture such as a RISC-based PE, pipelining involves partitioning the ISA into sub-functions which are common to all supported instructions. For example, all ALU instructions require either one or two input operands to be fetched from the register file before the actual ALU operation occurs. Once finished, the ALU result (if any) must be written back[3] to the register file for future use. Memory instructions follow a similar flow with the exception that memory write (store/STR) functions do not require a writeback operation since a memory store instruction is typically dispatched without any feedback. On the other hand, memory read instructions (load/LDR) fetch data from either external memory or cache (if any), with the 'fetched' data entering the microprocessor pipeline via the same port used to issue memory write instructions. RISC

[3]A memory operation which writes a value to memory is referred to as a store operation, while the writeback function refers to the process by which a value is written from the pipeline back to the register file.

microprocessor designs have tended to converge towards a number of common instructions, with more complex instructions such as multiply, count leading zero or population count added should the target application require such operations. While the ISA architecture employed within various designs shares a number of common traits, it remains possible to divide up the instruction pipeline in a number of ways. For example, the Texas Instruments MSP family of micro controllers utilise a single stage design [104], while the ARM7, ARM9 and ARM11 designs utilise a three, five and six stage pipeline respectively.

Within the DLX pipeline (Figure 2.8), instructions are first fetched from program memory before moving to the instruction decoder. Typically implemented as a logic array, the decoder translates an encoded instruction into the bit values needed to set the control lines within the CPU. Following the decode stage, the operands (if any) are fetched from the register base, the ALU output multiplexer is configured to select the correct result and the desired operation is processed. Following this execution stage, the memory stage involves accessing external memory, cache or memory mapped IO devices. Finally, the execution result or memory read is stored in the register file. In all, four edge-triggered pipeline registers are inserted to the design. As was noted previously, each pipeline stage can ideally operate independent of other stages, but a number of problems arise within strictly pipelined architectures. Firstly, not all instructions can be implemented within a single cycle. For example, operations such as binary multiplication require more complex combinational logic when compared to simple binary addition. Secondly, in order to maintain optimum performance the pipeline must be full at all times despite a number of hazard conditions which can occur within a pipelined design.

2.6.3 Pipeline Hazards

Given a PE executing application n on packet p, the time taken to process the packet is given by:

$$T_n(p) = \frac{N_{INSN} * CPI}{f_{clk}} \tag{2.4}$$

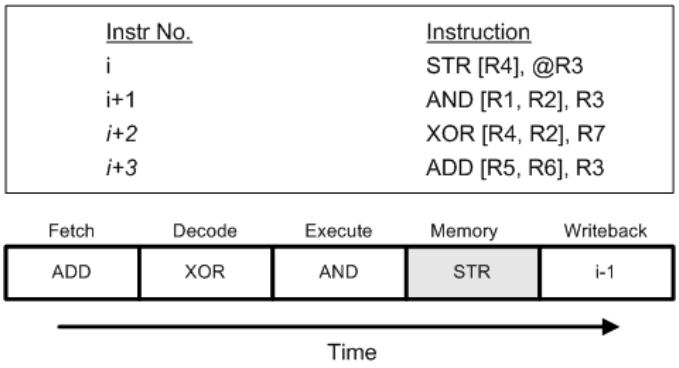


Figure 2.9: Structural Hazard Within Pipeline Processor

where N_{INSN} is the number of instructions in application n, f_{clk} is the clock speed of the PE and Cycles Per Instruction (CPI) is the average number of clock cycles required to resolve an instruction. To optimise performance the average number of clock cycles per instruction must be minimised, with a CPI=1 representing the ideal solution. The processor frequency can be driven higher by two factors, namely technology improvements and pipeline depth, but the average number of cycles per instruction can actually be increased by increasing the number of pipeline stages. As was noted previously, it is not always possible to *fit* every instruction along a given pipeline; for example the multiply and population count instructions both require long combinatorial circuits, significantly longer than the critical path associated with functions such as addition or bitwise shift. To accommodate these instructions, a mechanism for stalling the pipeline while the long operation completes is required. For operations such as multiply, the number of stall cycles is typically determined by the length of the multiplication with a 32x32-bit multiply requiring a higher number of stall cycles than a 16-bit operation. Each multiplication requires at least one stall cycle which will have the effect of increasing the average CPI, reducing performance. In addition to these ISA limitations, three possible conflicts also arise which can decrease performance by increasing the CPI.

2.6.3.1 Structural Hazard

The first type of hazard which is possible within a pipelined design is due to hardware limitations, typically imposed for reasons of cost. Assume the fetch and memory stages of the DLX pipeline share the same resource, as when a single port memory architecture

(Von-Neuman) is employed. A possible hazard occurs every time a memory operation is resolved in stage four of the pipeline. Using the assembly code snippet outlined in Figure 2.9 it is clear that during the current cycle, the processor will attempt to store the contents of register 4 (R4) in the address pointed to by register 3 (R3), while at the same time the fetch stage will need to load the ADD instruction into the pipeline to ensure maximum performance. The most obvious solution to this hazard is to separate the instruction and data memories, with two buses employed. Fortunately, for NP it is possible to implement separate instruction and data buses for all PEs without incurring much cost. In Chapter 4 a code analysis of NP applications is presented in which it is argued that the small program size seen in NP applications allows for a dual-port Harvard architecture to be employed, with the instruction memory coupled directly to each PE, removing the need for prefetch logic, branch target buffers or instruction caches.

2.6.3.2 Data Hazards

From the 5-stage pipeline outlined previously it can easily be seen that conflicts can arise between two instructions located in adjacent pipeline slots which utilise the same data operands. More specifically, a data conflict (hazard) can occur during a Read-After-Write (RAW) sequential lock. Consider the code segment outlined in Figure 2.10.

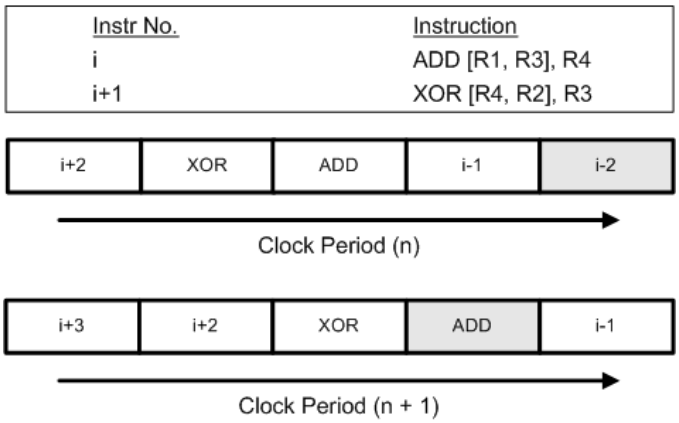


Figure 2.10: Data Hazard Within Pipeline Processor

In this case, instruction *i+1* will XOR register R4 and register R2, but since register 4 is due to be modified by instruction *i* the value sourced from the register file will be obsolete. Within hardware it is relatively easy to solve this RAW-type data conflict, with a

mechanism employed which allows data located in stages further ahead than the execution stage to be forwarded to the execution unit should the register indexes match. Although not applicable to scalar architectures, it should be noted that two other data hazard conditions are possible, with Write-After-Read (WAR) and Write-After-Write (WAW) conflicts possible due to data mismatches within a superscalar design.

2.6.4 Control Hazards

The last type of hazard which can occur in a pipelined processor is due to a change in program control flow. During normal program flow, the PC maintains the address of the next instruction to be fetched from memory. At the end of the cycle, the PC is incremented by some value, typically the byte wise width of the instruction word. As this instruction is latched into the decoder stage, the fetch stage will attempt to read another instruction word from program memory, assuming the program flow remains linear. A control hazard occurs when an instruction attempts to modify the program flow by jumping to another location via an alteration to the PC. Since the flow change cannot be evaluated until the decode stage at the earliest, it is not possible to know whether the next instruction to be fetched is located either at the next sequential address or the new target address, a value which is not available to the fetch stage until the next clock cycle. When programming a CPU in a high level language such as C, the most common flow control changes within a program are caused by the need to support functions such as *if-else*, *switch* or *loop* functions. In general, each of these operations can be translated into either a conditional branch (e.g. (*for(i=1;i≤10;i++)*) which may or may not modify the PC, or an unconditional branch (e.g. (*while(1)*) which will always modify the program counter.

In the case of conditional branches, the condition on which a branch is taken is determined by the status of the ALU status flags. Although different architectures will support a different number of status flags, the most common ALU status flags are the Zero flag (Z), Carry flag (C), Overflow flag (V) and Negative flag (N). Between these four flags it is possible to derive a large number of conditional operations, for example the common

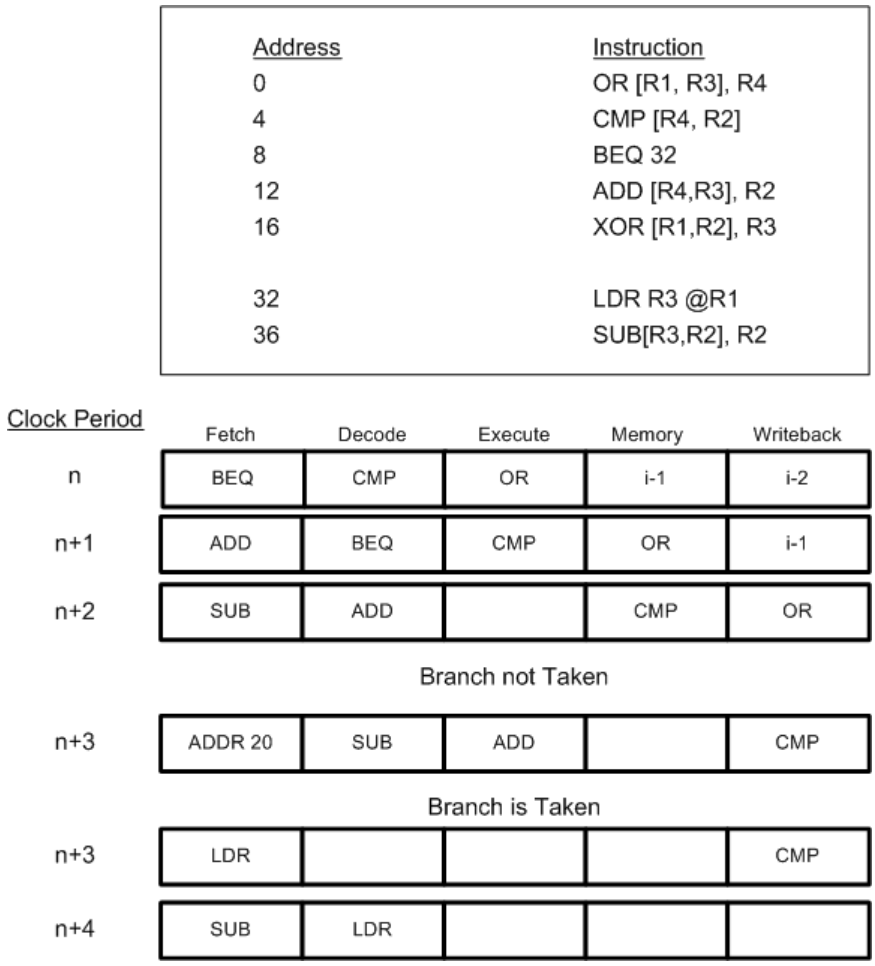


Figure 2.11: Branch Penalty Within Pipeline Processor

Branch if Equal (BEQ) instruction can be implemented by checking if the Z flag is set after a compare operation. While the four conditional flags outlined above are common within General Purpose Processor (GPP) architectures, it is possible to implement more *specific* branch instructions in order to aid conditional code generation. For example, the Intel IXP NP allows conditional operations to be evaluated on fields such as specific bits, specific bytes, the PE context or if an external signal has been asserted.

An example is presented in Figure 2.11 which illustrates a control hazard. In this case, the result of a branch decision is not known until the end of the execution stage. The CPU begins execution at address 0, sequentially fetching instructions from memory. On cycle *i* the fetch logic reads the *branch if equal to* instruction from address 8. On cycle *i+1* the fetch logic will attempt to read address 12 while the branch instruction will move one step ahead to the decode stage. Whether the conditional branch is taken will be determined

by the result of the previous compare instruction (CMP [R4,R2]). If the contents of the registers are equal the program flow should jump to address 32, else it should continue from the next address. It can be seen that, regardless of the conditional result, the branch instruction terminates at the decoder stage, with an empty slot propagating through the rest of the pipeline. For conditional branches which evaluate true (taken branches), the two misfetched instructions must be flushed from the pipeline since they are no longer required. This results in three empty slots propagating through the instruction pipeline, during which time no work is done. Since the empty slot due to the terminating branch instruction is common to taken, not-taken and always taken branches, the penalty associated with a branch operation (B_{pen}) is commonly referenced as the two slots which must be flushed from the pipeline. For unconditional operations, the branch can be evaluated in the decode stage requiring only one instruction to be flushed from the pipeline. When examining microprocessor performance, two factors become important when considering conditional hazards due to conditional branches. Firstly, additional pipeline stages located before the ALU flags will increase the branch penalty. Secondly, as the target application increases in size, the number of cycles lost due to branch instructions will also tend to increase.

Assuming branches are not taken, the branch penalty in terms of cycles for a branch instruction decoded as taken in stage *M* is for the *M-1* previous stages to be discarded. The slot occupied by the branch instruction is not considered since it is not possible to avoid this cost. The total penalty (τ_{pen}) can be calculated as:

$$\tau_{pen} = (\rho_{tk} * N_{br} * P_{tk}) \tag{2.5}$$

where ρ_{tk} is the average of ratio of taken branches, N_{br} is the total number of branch instructions in the program and P_{tk} is the penalty incurred during a taken branch.

Comparing control hazards to either structural or data conflicts, the primary difficulty is that, to date, it has not been possible to solve the control hazard problem fully. Instead, various techniques have been proposed which attempt to minimise this penalty by predicting if a branch is likely to be *taken* or *not-taken*. At its most basic level, these branch

prediction techniques attempt to utilise factors such as application layout, common programming paradigms or using run-time history as a means of guiding future branch predictions.

2.7 Techniques for Branch Prediction

This section presents an overview of various techniques used to mitigate the performance penalty associated with branch instructions in a pipelined architecture. Two methods have emerged for minimising the cost associated with branch operations. While the first method is broadly software based and is typically implemented via analysis at compilation, the second method utilises run-time history of the program as a means of guiding future decisions. It should be noted that not all static techniques must be implemented at compile time only, and various schemes have been proposed which are traditionally labelled static in nature but do utilise some hardware mechanisms.

2.7.1 Static Branch Prediction

With a few slight exceptions, static branch prediction can be broadly defined as *at-compilation* based techniques which may utilise; profile statistics, path information or general heuristics to determine likely branch outcomes. An exception to this *fully* software based definition can be seen in the forward not-taken/ backward taken scheme. Program analysis in [105] by Smith found that, for loop intensive operations, those branch operations jumping backward in code will typically be taken, while those branch operations which are forward pointing are less likely to be evaluated as true. While the compiler can reorganise the application code to reflect this information, the micro-architecture must be designed to reflect this, with any prediction logic comparing the current program counter to the branch target address. A coarsely grained prediction mechanism, the forward taken/backward not taken scheme clearly mispredicts a large number of conditions. For example, the last iteration of every loop is predicted incorrectly while a conditional jump to a subroutine requires extensive code reordering. An extension of such hybrid-static techniques

might be to encode additional information in the instruction word at compile time. The hardware is then redesigned so that a branch prediction decision takes into account not only the direction of the branch but also whether the branch was hinted as likely to be taken or not at compile time [106].

Referring back to the DLX pipeline it can be seen that the next instruction to be fetched after a branch instruction is purely speculative since it is unknown whether the branch will be evaluated true or false and whether the pipeline must therefore be flushed or not. A simple solution to this wasted slot might be to move an instruction which is unrelated to either branch decision but would be required later to this slot. An example of such a case might be where a variable is set after a conditional call to a subroutine. Although efficient in that no additional hardware is required and no pipeline stages are wasted, there are a number of difficulties associated with filling every post branch delay slot with an independent instruction. Firstly, there must always be an independent instruction which can be positioned in the delay slot. Since most applications are sequential in nature this is not always possible. Secondly, as the number of pipeline stages increases, the number of delay slots also increases, meaning multiple independent instructions must be relocated. Finally, the compiler must take into account these relocations when allocating register space to variables. In the event of no instruction being available, the compiler will be forced to insert a No-Operation (NOP) into the delay slot. In [107] it was found that branch instructions comprised 10.96% of all instructions executed for the SPEC benchmark, while the injected NOP operations comprised 8% of the total instruction workload.

While both of the solutions outlined above are simple to implement, more complex static analysis methodologies are possible. In general, static techniques typically fall into one of two categories, *profile*-based static prediction, which attempts to extract prediction information from sample runs of the target program, and *program*-based prediction, which maps heuristic rules to the target program. Fisher and Freudenberger found that by subjecting a target program to a number of previous runs, it is possible to deduce the likely direction of branches regardless of the dataset [108]. Such *profile* (or *path*) based tech-

niques can be expanded to include *path* information based on logical correlation [109]. The difficulty with such techniques is the time required to generate the input traces, profile the target application and finally recompile the modified target program to include the new branch information. The work in [110] highlighted some of the performance issues related to the original static correlation method outlined by Young et al [109]. Reducing the need for recompilation, other static methodologies have encoded the profile information in a *likely* bit within the branch instruction. This likely bit is then used at run-time by hardware to guide any branch prediction. At a more fundamental level, the problem with *profile* based static prediction is how to generate the trace dataset used to profile the target application. In [111] it was found that real profiles provide significantly better coverage than either estimated or random traces. From an NP perspective, the limitation with regards to the *profile*-based technique is how variations in network traffic would affect any predictions. Consider a NP router running packet metering and IPv4 forwarding. Over a short period of time the metering algorithm must handle short bursts of traffic [112]. Over a longer period, say 24 hours, the profile of the metering algorithm would adjust to reflect the periods of under-utilisation. At the same time, the execution trace for the forwarding application would adjust to routing table changes. On the other hand, the fact that the majority of packets must pass verification for the network to remain viable should make *profile*-based techniques applicable to NP systems.

Another common method of providing static branch prediction is via *program*-based correlation techniques. As was noted previously, work by Smith [105] and McFarling [106] found that common programming idioms can be used to detect likely conditional outcomes. In [113] and [114] a number of more structured heuristic rules were defined which could be used for static branch correlation. The basic methodology employed within a *program*-based technique is to detect whether a branch operation is likely to be taken, based on the program structure. For example, routines required for error processing are rarely called and it can therefore be assumed that any conditional call to an error sub-routine would rarely be taken. Calder et al. proposed a more global framework for *program* based prediction, in which heuristic rules are obtained from an analysis of the

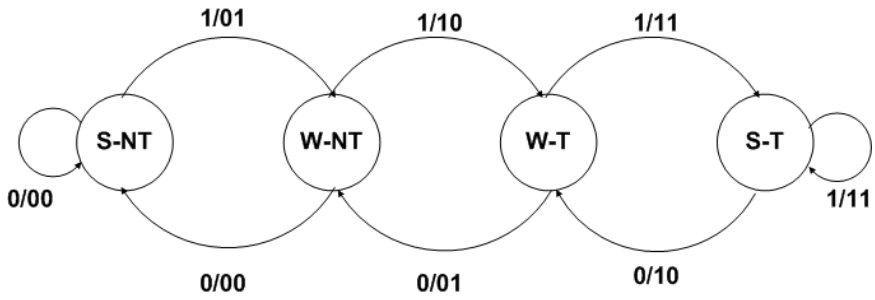


Figure 2.12: Sample 2-Bit State Transitions for Branch Predictor

large programming body currently deployed [115]. When compared to *path*-based techniques, the primary advantage of this method is the removal of any testing and simulation steps when profile traces are extracted. Instead, the *program* heuristics can be incorporated at compile time [116].

Within general purpose processing static prediction techniques have not provided the prediction rates required for modern microprocessors. Neither *profile* nor *program* frameworks obtained prediction rates in excess of 90%, but both methods do provide a means of optimising object code during compilation, e.g. static prediction provides a method of removing redundant paths [117], and is commonly implemented at compiler time.

2.7.2 Dynamic Branch Prediction

With static based prediction techniques failing to take into account future changes in the program flow and therefore branch behaviour, dynamic branch prediction attempts to utilise the run-time execution history when evaluating whether a branch operation will likely be taken or not. Typically a hardware based solution, the premise of dynamic history is that, by assigning a finite number of states to each branch operation within the application, it is possible to predict future directions of that branch based on the current branch state value, a value which in turn was determined by previous evaluations of the specific branch operation. The most common method is to assign each branch to a 2-bit saturating counter, with the branch instruction transitioning between four possible states. In Figure 2.12 a sample transition scheme is shown for a 2-bit branch predictor.

For each branch operation mapped to a saturating counter, a prediction can be obtained

from the most significant bit of the counter state. For each not taken branch the counter is decremented by one (to a minimum of 0), while a taken branch is recorded by incrementing the counter logic (to a maximum of 3). Moving from left to right, the states are assigned Strongly-Not-Taken (S-NT), Weakly-Not-Taken (W-NT), Weakly-Taken (W-T) and finally Strongly-Taken (S-T). At initialisation the counters are set to a random distribution of weakly taken/not-taken, with the initial branch operations providing training information to the prediction logic. Once fully trained, an ideal 2-bit system will predict correctly for heavily taken branches all of the time except for the last iteration, at which point the loop will break from its runtime history. It should be noted that Figure 2.12 illustrates just one of various configurations, with other state transitions possible [118]. All configurations involve some degree of trade-off with the original scheme mispredicting both the last and first iteration of a loop, while the scheme outlined above would have mispredicted two branches either taken or not taken when the counter is in the corresponding saturated state.

```
for(i=0;i<5;i++) {
    if(i == 3)
        do task A;
}
```

Listing 2.1: Sample Branch

To demonstrate the operation of a 2-bit scheme, consider the code snippet shown in Listing 2.1. Assume the *for* and *if* statements map to separate branch counters (i.e. no branch interference). Typically, prediction locations are initialised as weakly taken or not taken. In Table 2.4 the branch trace is outlined for all iterations of the *for* loop and *if* statement when the predictor is initialised as weakly not taken (01).

The second and sixth columns within the table specify if the branch will be taken (Yes) or not (No), while the third and seventh columns refer to the predicted decision for both branch operations. The updated state (after the current branch has been accounted for) is shown in the ST_{for} and ST_{if} columns. When the predicted (PR) and taken (TK)

Table 2.4: Prediction Performance

Iteration	**TK**$_{for}$	**PR**$_{for}$	**ST**$_{for}$	**MS**$_{for}$	**TK**$_{if}$	**PR**$_{if}$	**ST**$_{if}$	**MS**$_{if}$
i=0	Yes	No	”10”	Yes	No	No	”00”	No
i=1	Yes	Yes	”11”	No	No	No	”00”	No
i=2	Yes	Yes	”11”	No	No	No	”00”	No
i=3	Yes	Yes	”11”	No	Yes	No	”01”	Yes
i=4	Yes	Yes	”11”	No	No	No	”00”	No
i=5	No	Yes	”10”	Yes	N/A	N/A	N/A	N/A

columns mismatch it is clear that the branch predictor has mispredicted the direction of the branch. Tracing the first iteration of the code segment it is clear that the *for* loop will be mispredicted since it has been initialised as weakly not taken (01) while the *if* statement will predict correctly. During the second iteration the state for the *for* loop has incremented to weakly taken, correctly predicting the second iteration, while the state for the *if* operation will decrement to strongly not taken, also correctly predicting the direction of the branch. Stepping forward to the final iteration of the loop it is clear that the *for* loop will be saturated, mispredicting the exit condition of the loop. Since the loop branch operation would be placed after the *if* statement in the assembled code, the last iteration of the loop does not require the *if* to be evaluated. In total, 11 branch operations are encountered during execution of the code segment, with 3 branches mispredicting. The overall hit rate is therefore 72%.

With modern applications comprising multiple branch operations the number of 2-bit entries available for each branch operation must be arranged in a table format, commonly referred to as the Pattern History Table (PHT). Since only one PHT entry is accessed at a time it is possible to significantly lower the area cost of the prediction architecture by implementing only a single 2-bit saturating counter along with a series of *n*-word by 2-bit memory entries, where *n* is the number of entries in the PHT. Outlined in Figure 2.13, this method allows the PHT area to be reduced to a number of 6 transistor SRAM bits along with the control logic required to access the SRAM locations and the combinational logic needed to increment and decrement the PHT entries. With multiple branches mapped to different locations but requiring single cycle access, the most challenging aspect of

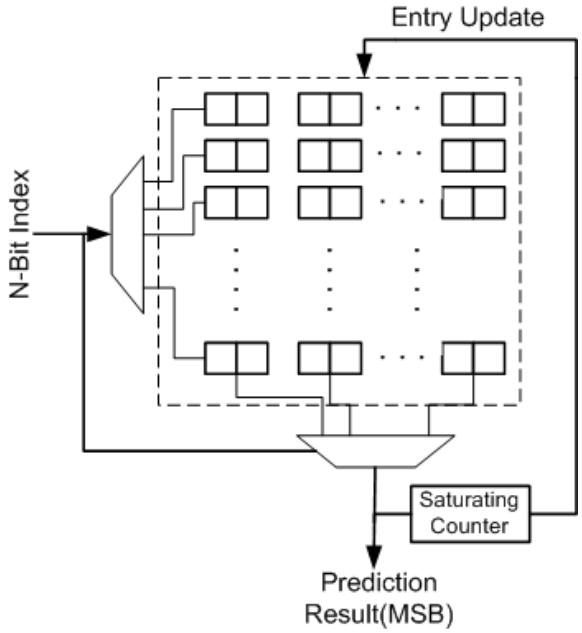


Figure 2.13: 2-Bit Predictor Table in SRAM

a dynamic predictor is how these branch operations can be mapped to the PHT table without multiple branches being located at the same entry. Consider an application which involves *s* separate branch operations, each of which is mapped to a separate entry in a *k*-entry PHT. The load factor for the application is therefore given by:

$$\alpha = \frac{s}{k} \tag{2.6}$$

Similar to any linear search structure, the mapping function employed in branch predictors is typically a hash type function which translates an input bit stream into a shorter bit stream in the region $0, 1, 2...k-1$. Since a branch predictor is indexed on a per branch basis, the Program Counter (PC) provides the most obvious input key through which the PHT is indexed. The task of the predictor hash function is therefore to reduce the length of the PC address to another address within the PHT address space, while also providing a uniform distribution of addresses across the PHT table. The original dynamic predictor outlined in [119] used *k*-bits of Program Counter as an index to access a 2^k PHT. Since it suffers from high collision rates due to the limited distribution provided by the program counter, a directly mapped structure has been replaced by dynamic predictors indexed using a Global History Register (GHR). Commonly referred to as a two-level system [118]

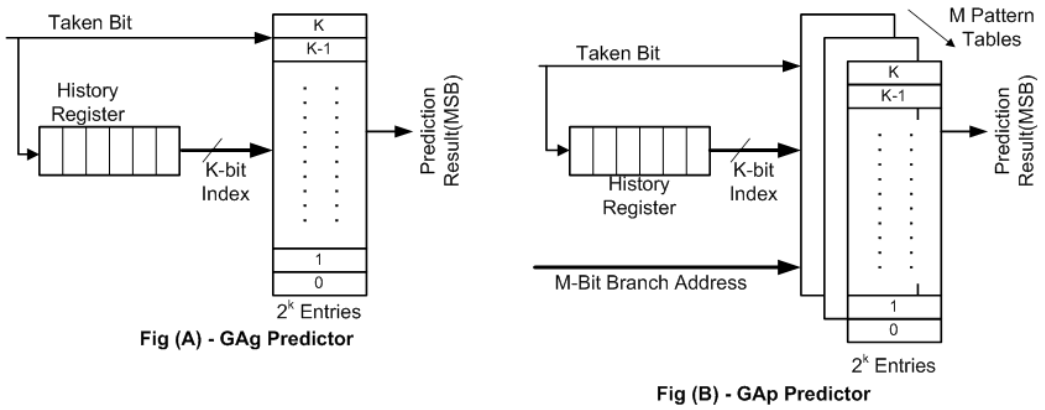


Figure 2.14: Global History Prediction Schemes

[120], there are a number of different configurations possible within such a scheme, each of which is identified using the naming system outlined by Yeh et al. [118].

In the first case the branch taken bit is fed into a *k*-bit global history shift register, labelled the **GAg** predictor. On addition of a new bit, the *k*-bit index within GHR is used to access a 2^k PHT. As is clear from Figure 2.14(A), the area of this configuration can be estimated at $(6 * 2 * 2^k) + (k * 16)$, assuming 6-Transistor SRAM cells and 16-transistor edge-triggered register bits. Since the history register is global to all branch operations, the k-bits of the index change rapidly, allowing a more uniform distribution of the branches across the PHT. With applications typically following a repetitive structure, repeated branches are assumed to be indexed in the same order each time. An obvious flaw with such a configuration is where a very intermittent branch is added to the last *k* branch operations which occur regularly. In this case, the lack of any Per Address (PA) history has limited the performance by randomising the branch history at a global level only. Since the GHR heavily reduces the input space to a k-entry linear search space, interference between branches significantly reduced prediction performance. An easy solution to such a problem is to parallelise the PHT tables as shown in Figure 2.14(B). In this case *m*-bits of the branch address (Program Counter) are added to the *k*-bit GHR to index 2^m separate PHT. The area estimation for such an architecture is approximately $(6 * 2 * 2^k * 2^m) + (k * 16)$.

In order to minimise the effect of the global history on the current branch a more efficient solution than the GAp architecture outlined above is to parallelise the global history

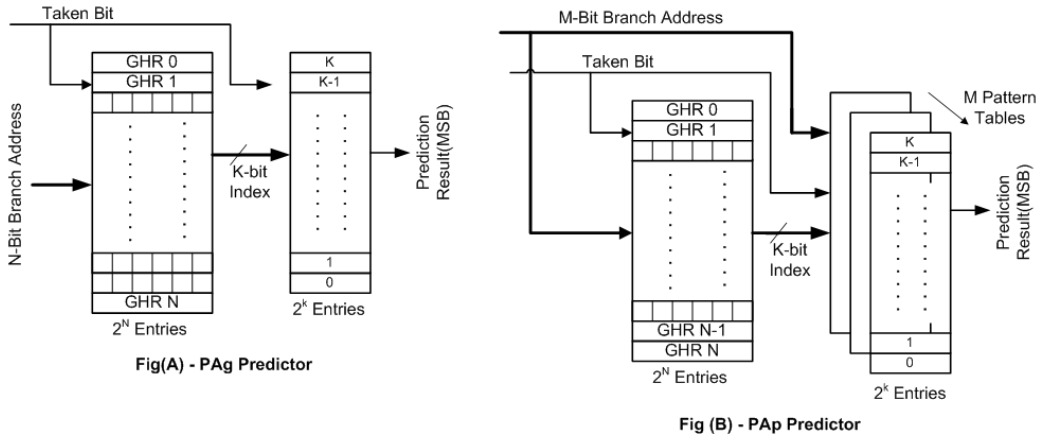


Figure 2.15: Per Address Dynamic Prediction Schemes

via the branch address, allowing 2^n separate GHRs to be indexed via n-bits of the program counter. Shown in Figure 2.15(A), the *PAg* predictor alters the structure so that the GHR represents the the history of the last *k* occurrences of the branch operations indexed by the *n*-bits of the program counter. Similar to the GAg scheme, the primary difficulty with such an architecture is that by reducing the program counter to 2^n PHT space, collisions between branch addresses remain common, with branch interferences destroying both the GHR and therefore the PHT entries indexed by the GHR. Similar to the PHT architecture outlined in Figure 2.13, a method of implementing a parallel GHR is to design the sequential logic as an SRAM array of *n*k* bits with a single combinational block for the shift function. The area of the predictor is therefore $(k * 2^n * 6) + (2 * 6 * 2^k)$ transistors. The final configuration of the 2 level predictor is to employ both a per address GHR table as well as a per address PHT structure. Commonly referenced as the **PAp** predictor, the structure is outlined in Figure 2.15(B). The number of branch address bits utilised is expanded out to *m* + *n* with *n*-bits of the address used to index the GHR table while the *m* bits allow multiple PHT tables to be accessed. The transistor cost for such an architecture is given by $(k * 2^n * 6) + (2 * 6 * 2^k * 2^m)$

Summarising the above architectures it is clear that the trade-offs within branch prediction can be described as how the PHT table size can be minimised without introducing conflicts within PHT entries. While an increase in number of available PHT entries will reduce the number of branch conflicts, the load factor will be decreased for a fixed number of branch instructions. Parallelising the structure either via multiple PHTs or through

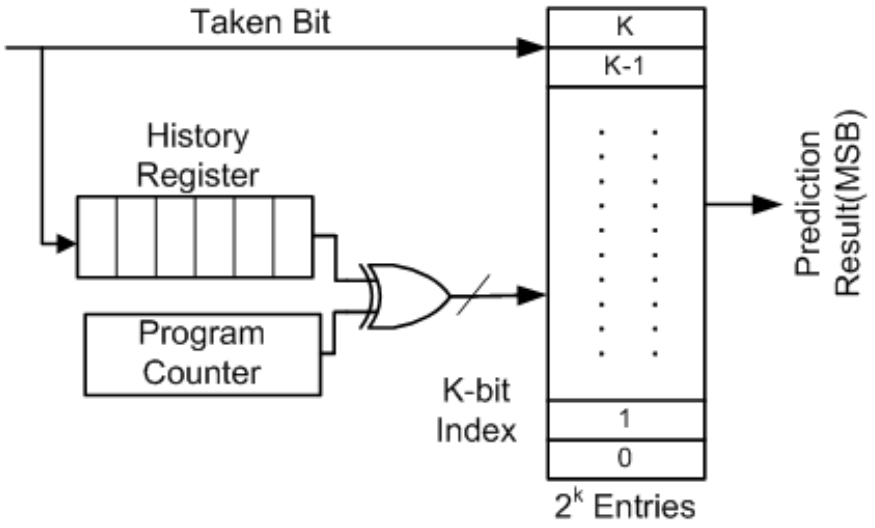


Figure 2.16: Gshare Predictor

a Per-Address GHR can reduce some of this interference but increases the area cost by a factor of 2^m, $k*2^n$, and $k*2^n+2^m$ respectively. In addition to these configurations, Pan et al. have also proposed a correlating dynamic predictor which is similar to those outlined above [121], while Macfarling proposes the *gshare* and combinational predictors [122]. In the case of a combinational predictor, two differing predictors are used to first generate separate prediction results. Both of these predictions are then fed into an output predictor which decides which of the first predictors should be used. A variation of this architecture is proposed in [123]. In addition to these combining architectures, McFarling found that a simple XOR operation between the program counter and GHR can significantly improve prediction rates. Commonly referred to as the *gshare* predictor, it is shown in Figure 2.16.

Other dynamic predictors include the *Agree* predictor [124] which attaches a biasing bit to each branch instruction. The two bit counters are then used to compare if the branch will go in the direction indicated by the bias bit. The two must therefore agree in order for a prediction to be used, otherwise the prediction defaults into some fallback position. For NP platforms a number of difficulties with this architecture are apparent. Firstly, the bias bit is attached to the branch instruction via either a branch target buffer or instruction cache, but neither of these components are required within a PE. With a small application kernel, there is no need for PEs to calculate the branch target, which must be buffered. Similarly, without an instruction cache, the bias bit must be made static at compilation.

A second difficultly is that, while the *Agree* predictor does improve prediction rates, the need for a large pattern history makes it an expensive solution. For example, the *xslip* algorithm requires an 8K-entry table to achieve prediction rates in excess of 95%. In [125] the bi-mode predictor was proposed, splitting the PHT into three smaller tables, a not-taken prediction table, a taken prediction table and a choice table. Similar to the *Agree* system, the bi-mode predictor improves performance when compared to traditional *gshare* architectures. The difficulty for an NP system is that a large predictor is not feasible for cost reasons. The minimum size examined in [125] is 0.5KB or 2K-entries per PHT$(512 * \frac{8}{2})$, with two PHTs employed in parallel. Another method of improving dynamic predictor performance through reductions in the interference rate was proposed in [126]. Performing better at small table sizes than previous solutions, the *YAGS* predictor assigned a tag to each branch which was cached in either a taken or not-taken cache. It suffers from similar limitations to the bi-mode scheme, in that large predictor sizes are required for prediction rates above 95% ($\sim$ 10 K-Bytes or $\sim$80 K-bits). It should be noted that the *Agree*, *Bi-Mode* and *YAGS* schemes all attempt to minimise the effect of destructive aliasing, with such behaviour more noticeable in larger general purpose applications. With NP applications following a much smaller and tightly bounded framework it should be possible to optimise prediction architecture in ways other than by simply improving how branches are mapped to the PHTs.

2.8 Conclusions

The trend within NP design is to support more intelligent services while simultaneously increasing the number of packets processed by the NP. Higher bandwidth connections reduce the amount of time available for each packet to be processed, while the need for functions such as metering or filtering is increasing the complexity of NP-based applications. Modern routers must perform tasks such as packet classification, packet inspection and packet encryption along with the traditional functions such as IP forwarding and flow metering. No single architecture has yet emerged which is optimised for all applications, with certain functions better suited to parallel implementations while others provide better

performance when partitioned into pipeline stages.

A number of methods are available for improving NP performance. The first method is to increase the number of process engines employed on-chip. Additional PEs allow the programmability to be retained but the degree of parallelism that can be deployed is limited by factors such as power consumption, device contention and compiler issues. With memory access speed failing to match the performance increments seen in microprocessor design it is clear that the implementation of additional PEs is unlikely to provide a long term solution. The second method involves offloading computationally intensive tasks to dedicated accelerators, either located on-chip or networked externally to the NP. Sacrificing flexibility for performance, dedicated logic allows optimum algorithm performance. With network applications and demand in a state of flux, it remains unclear whether hardware offloading is justified since it may not possible to simply upgrade existing hardware, increasing cost while also slowing the evolution of Internet based functions.

Another method of improving performance is through micro-architectural techniques which target the PEs. Traditionally employing a simple RISC architecture, it is possible to tailor the underlying architecture towards NP applications without sacrificing the underlying benefits of a programmable system. These techniques include methods such as multi-threading, flow-based cache, network specific instructions or by implementing deeply pipelined PE architectures.

As with all digital systems, a deeper pipeline allows additional performance to be extracted by increasing the maximum clock speed of the architecture. This performance gain can only be guaranteed if the pipeline is kept full at all times, a challenge when branch operations are taken into account. Various techniques have been proposed to mitigate this branch penalty and the primary focus of this thesis is to analyse and explore this topic within NP systems. The exploration and analysis of micro-architectural techniques present a challenge to NP researchers since it requires methods for evaluating the performance of NP systems. The most common and powerful of these methods is to use a simulation model for the system under consideration, a topic which will be considered in the next chapter.

CHAPTER 3

Performance Evaluation Methods for Network Processors

3.1 Overview

The survey of NP architectures presented in the previous chapter highlighted that within the NP domain no single optimal architecture has emerged, unlike the GPP domain. Various architectures have been proposed in both commercial and academic research. There are however a number of common components from which the majority of NP architectures are constructed. Clearly the PE-based data-plane processing array represents one such component. Other components include; a multi-channel memory hierarchy and an on-chip network transceiver to connect the NP to the network and switching fabric, while possibly including additional hardware blocks for complex tasks such as packet encryption. With a modern NP SoC comprising multiple modules, a challenge to NP research is how to model, simulate and evaluate the NP design space. When considering topics such as workload analysis, memory behaviour or branch prediction this challenge becomes a significant problem since any research model should encompass the entire NP design space. Comparing NPs to more general purpose systems the deficiency in terms of an available simulation model can be clearly seen. Within general purpose processing RISC simulators, such as SimpleScalar [127], allow various RISC architectures (ARM, Sparc, Superscalar) to be selected as the basis for a functional simulation. For micro-architectural aspects such as power optimisation and cache performance, tools such as

Wattch [128] and Dinero-IV [129] can be used for research and development. Additionally, it is common for microprocessor and microcontroller manufacturers to release a cycle accurate model of their underlying hardware. Within the NP domain, however, there is no unified methodology by which an NP architecture can be efficiently modelled. Previous NP research has tended to utilise either a general purpose framework or a commercial platform. Unfortunately, neither of these methods is sufficient when performing a thorough examination of common NP aspects such as hardware acceleration, PE parallelism or micro-architectural techniques such as thread-level parallelism or branch prediction.

In this chapter a discussion of the problems and challenges regarding NP modelling and performance evaluation is presented. In addition to a survey of existing modelling techniques, a number of NP specific performance evaluation metrics are introduced which are used in future chapters when evaluating NP and PE performance.

3.2 Simulation and Modelling of NP Architectures

Unlike GPP which has largely settled on a multi-core Complex Instruction Set Computer (CISC) architecture for desktop computing while using multicore RISC platforms in mobile and embedded solutions, no single optimum NP architecture has emerged from either the commercial or academic domains. When modelling such a system, the challenge is therefore how to encapsulate the wide number of configurations possible (despite using similar sub-modules), while also developing a simulation model which allows accurate information to be extracted for analysis.

Within computer architectural modelling and simulation there are a number of differing methodologies which allow microprocessor systems to be modelled. While some of these models focus on describing the system via a number of steady state equations, other software based models allow greater flexibility by simulating the underlying hardware as either a functional block or cycle-accurate models. The NP simulator developed by the author and proposed in this chapter, SimNP, follows the traditional functional simulation framework. Within a functional model, each of the sub-modules are evaluated at a functional level only, guaranteeing functional performance but lacking cycle accu-

rate statistics. Before outlining the SimNP simulator (Section 4.2), a brief preliminary background on system modelling is presented.

3.2.1 Mathematical Models

With complex SoC designs requiring a large outlay of resources and time, a method of evaluating performance for various configurations before the design process begins is sometimes required. Constructed to model steady state operation, stochastic mathematical models represent the most common methodology. In turn, these stochastic models can be separated into two sub-methodologies which are outlined below.

3.2.1.1 Queue Model

Queue-based models allow the delays, loads and probabilities within a system to be approximated without a detailed simulation framework. Once a queue model has been developed, analysis of the queue and service nodes within the system can be used to identify potential bottlenecks and contention. For example, consider the system outlined in Figure 3.1.

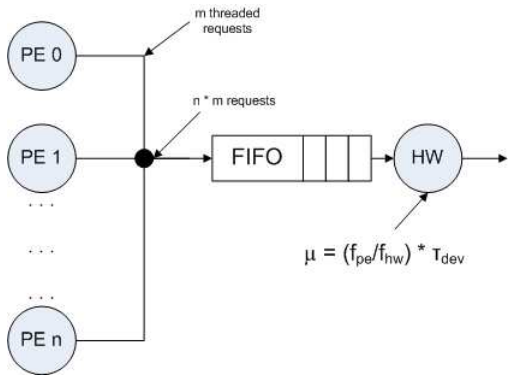


Figure 3.1: Queue Model

Each of the n PEs access a shared device such as a memory module, interface unit or hardware accelerator. Each PE hosts m threads, with each thread performing the same task. The PE array operates at f_{pe}, while the shared device operates at f_{hw} and takes τ_{hw} device cycles to complete one operation. To access the hardware block, each thread issues one command to the hardware block. In a steady state, the total PE request process is the

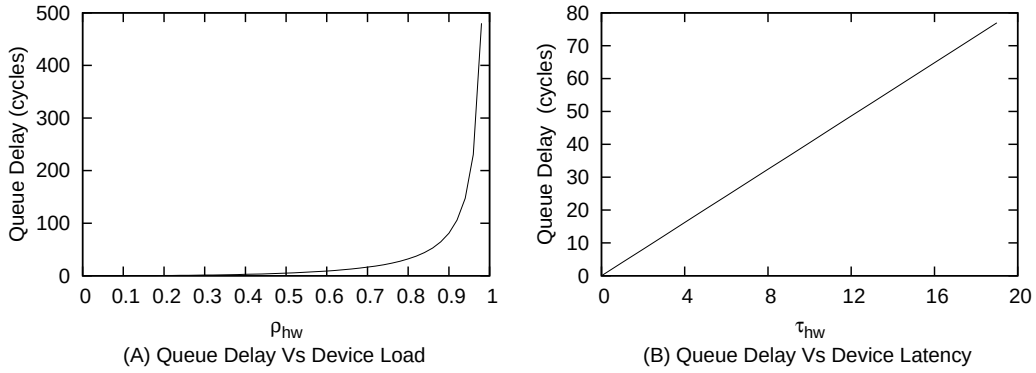
non-deterministic sum of *n* Poisson processes, with an arrival rate, λ, of *m* * *n*. The deterministic service time for each command is given by Equation 3.1.

$$\mu = \frac{f_{pe}}{f_{hw}} * \tau_{dev} \tag{3.1}$$

Following an M/D/1 queueing model [130], the total delay associated with the hardware accelerator can therefore be calculated using equation 3.2, in which ρ_{hw} is the utilisation rate associated with the hardware block ($\rho = \frac{\lambda}{\mu}$).

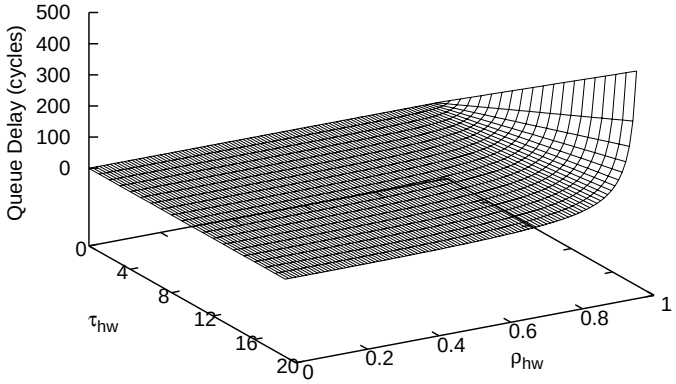
$$t_{queue} = \frac{\rho_{hw}^2}{2(1-\rho_{hw})} * \frac{clk_{pe}}{clk_{hw}} * \tau_{dev} \tag{3.2}$$

The queue model can therefore be examined at an abstract level for system performance, most notably for sensitivity to other system variables. For example, figure 3.2 presents the system delay as the load, access latency and hardware time is varied.



(A) Queue Delay Vs Device Load

(B) Queue Delay Vs Device Latency



(C) Queue Delay Vs Device Load Vs Device Latency

Figure 3.2: Simulation Results for Shared Hardware Block

In figure 3.2(A) it can be seen that the delay associated with accessing the hardware accelerator scales exponentially as the hardware load is increased, while in figure 3.2(B) it can be seen that changes in the service time, due to additional latency, will result in a linearly increasing delay. The non-linear response associated with device load presents a trade-off during system design. Such an open queue model allows rapid development and evaluation. Within general purpose processing these queue models have been used as a mechanism for performing highly abstract analysis of multiprocessor systems [131], [132], [133] and [134]. Queue models do, however, suffer from a number of problems. Firstly, a number of assumptions regarding task complexity and inter process communication are typically required [131], [132]. While Bucher and Calahan found that an open-queue model will overestimate delays by up to 10% [134], Tsuei and Vernon found that that a queue model developed specifically for an underlying architecture had an error of 9% when compared to a software model of the same architecture [133].

Wolf et al. presented an analytical performance model for an NP, with a queue model employed to represent data transfers between the PEs and system memory in [135]. The difficulty remains that, in order to extract meaningful data from the model, it was required to implement applications and algorithms on an architecture before performing any analysis, presupposing an underlying architecture. Furthermore, it was assumed that the system bus connected PEs and external DRAM via a cache, with no additional data generating bus or memory requests. In Chapter 2 it was seen that it is more common for an NP architecture to employ both fast, expensive SRAM-based control memory and slow, cheap DRAM-based packet memory.

3.2.1.2 Petri Net

A second mathematical framework for modelling is to use a Petri Net modelling framework, which provides a methodology by which a discrete system can be represented and analysed [136]. Examining transitions between concurrent systems, the Petri Net framework is well suited to parallel systems such as NP architectures. Within the NP domain, research in [137] examined the accuracy of Petri Net modelling when applied to the Intel

IXP architecture of NPs. It provides similar results to a cycle-accurate simulation but with a significant number of assumptions included. For example, a fixed line rate, fixed packet size and fixed memory access time are all assumed. Furthermore, it is unclear how the accuracy of a Petri Net could be improved without obtaining more accurate timing information, a process which would require applications to be implemented on the target platform.

3.2.2 Architectural Simulators

Operating at a lower abstraction level than an analytical model, functional simulations provide a mechanism for estimating the performance of various architectural configurations. With SoC-based NPs embedding more functions on chip, the number of underlying configurations can expand greatly, with researchers requiring fast methods of examining each configuration. In general, functional simulators can operate as either a full system simulation, in which only the overall functionality is verified, or as a transaction-based modular framework, in which certain components might be modelled at a higher abstraction level to more fundamental components such as the bus network. While a system level functional simulation might define how the data flows between simulated blocks, a transaction level framework decouples the communication between blocks and the functionality provided by the simulated blocks.

3.2.2.1 System Simulators

To date, a number of functional simulation frameworks have been proposed for general purpose architectures. Firstly, the widely used SimpleScalar framework [127] provides a means of examining application profile, branch prediction, instruction level parallelism, cache performance and micro-op performance. Capable of being configured to target either an ALPHA [138] or ARM [23] architecture, the advantage of SimpleScalar when compared to other platforms is the ease with which changes can be incorporated. While SimpleScalar suffers from a lack of micro-architectural accuracy, the MASE toolset presented in [139] was able to build on the SimpleScalar framework. For shared memory

multiprocessors the RSIM [140] simulator provides models, with processors capable of exploiting ILP. To rectify the lack of memory timing precision found in a high level functional simulators, DRAMsim [141] allows cycle accurate memory information to be extracted, while complete general purpose systems can be modelled via Simics [142] and SimOS [143].

Within the networking domain, Raswammey et al. [144] present an extension of SimpleScalar called PacketBench which provides a framework for benchmarking network applications. Defining where variables are stored, it allows memory accesses to be segmented into control and packet memory, while providing workload statistics on a per packet basis. When compared to previous workload and benchmark analysis [100], [91] and [90], it provides a mechanism for ensuring removal of any simulation 'wrapper' analysis from the results. Unmodified SimpleScalar based frameworks suffer from a number of limitations. Utilising only a single microprocessor, memory and bus contention issues are ignored. Also, segmenting variables into either packet or control regions represents a coarse grain solution, with certain architectures providing little register storage space and therefore requiring additional memory operations. RISC architectures such as ARM heavily utilise load and store instructions during stack operations. It is expected that these would be low (or zero) latency operations which must be separated from longer shared memory operations. In [145], Bhuyan and Wang present a single processor IP forwarding model derived from RSIM. In [146], Suryanarayanan et al. present a component network simulator called ComNetSim which models a Cisco Toaster NP, providing a functional simulator which is defined by the execution of applications being modelled, limiting the ability to compare different applications providing the same service.

3.2.2.2 Transaction Level Simulators

While higher level simulations such as the frameworks outlined above allow rapid development and analysis, the need to specify the communication framework between simulated blocks limits the flexibility and use of such models. Following a SoC design philosophy, a full system simulation of an entire NP must account for the various blocks found

within a modern Integrated Circuit (IC). While RSIM can be used to model the communication between the the processing array and memory, no solution outlined above can be configured to mimic the entire architecture.

As a solution to this, transactional simulators attempt to separate each block into various sub-modules. By defining each sub-module in terms of how it communicates, it is possible to model the functionality of the underlying block at a higher level of abstraction. The primary advantage of this transaction level approach is the ability to trade abstraction and flexibility against precision. In [147], Kohler el al. provide a modular mechanism for specifying processing functionality, but it was limited in the ability to evaluate specific architectures. StepNP [148] proposed a transaction based simulator which builds on the Click router [147] to allow system level exploration but is currently not available for public download. Various other SystemC transaction level simulators have been proposed [149] and [150]. The SimNP platform follows a transaction level framework, with NP based sub-modules connected via a centralised bus network.

3.2.2.3 Cycle Accurate Simulations

Commonly developed to accompany commercial products, cycle accurate simulation models represent the most accurate simulation models. Implemented to mimic an underlying architecture, cycle accurate simulation favours accuracy over development time. Complex to design and implement, analysis using a cycle accurate simulator requires the user to have a deep knowledge of the underlying architecture. Within the GPP domain cycle accurate models remain common. For NP platforms, the framework presented in [151] represents the only open source NP simulator currently available. Designed to mimic the Intel IXP12xx and IXP24xx architectures, NePSim models the PEs, bus and memory hierarchy and the interface unit. Compared with IXP1200s own cycle accurate architectural simulator, NePSim matches the Intel IXP platform to within 1% of packet throughput and 6% of the processing time. With the implementation developed to accurately model the IXP family, use of NePSim is limited by the fact that the IXP compiler is relatively underdeveloped when compared to existing general purpose solutions. Examining pub-

lished research, the open source aspect of NePSim does not appear to have been used by many researchers, with large numbers of research papers continuing to utilise Intel's own simulator [96][76][78][152].

The primary limitation to cycle-accurate simulators is that while simulators such as SimpleScalar can be programmed in high level languages such as C or C++, cycle accurate platforms such as NePSim and Intel SDK require extensive hardware knowledge of the underlying platform. Analysis by Peter Reiher in [153] found that the majority of IXP programming must be done in assembly code, with detailed architectural knowledge needed to optimise application performance. Frameworks outlined in [154] and [155] propose methods of improving the programmability of the Intel architecture but compiler limitations continue to limit the ability to fully utilise cycle accurate simulators in NP research. The four applications shipped within the Intel SDK represent the most common applications to be cited by researchers within the NP domain.

3.2.2.4 NP Programmability Challenges to System Modelling

A major limitation to accurately and rapidly modelling NP systems can be summarised as the lack of a fully developed programming framework for NP systems. Unlike GPP systems which have various programming languages, each targeting a specific level of abstraction, the NP platform is typically programmed in low level languages such as assembler or C. When accounting for functions such as thread control, inter-process communication and load-balancing, NP applications can quickly become complex to either develop or maintain. Within the NP domain a number of researchers have proposed methods of improving NP programmability. Both Lee [154] and Shah et al. [155] propose frameworks which aim to improve the programmability of NP systems. In [96], Ostler et al. describe heuristic algorithms and methodologies for mapping applications to a multi-processor, multi-threaded, NP (Intel IXP). In [96] the Shangri-La compiler was proposed as a means of generating binary images from a C-like language, again targeting the Intel IXP platform. Li et al. propose techniques to allow automatic partitioning across a pipelined NP [156] while in [157] a transformation method to automatically inject multi-threading

and multi-processing was described. Meijer et al. investigate methods for automatically partitioning stream-applications when implemented on an IXP platform [158].

3.3 Performance Metrics for NP Architectures

Along with the difficulties of constructing a model to reflect the modern NP design space is the challenge in how to accurately (and fairly) evaluate an NP architecture. In this section a brief discussion of performance evaluation metrics and methods is presented. These methods include analysis methods which are common to GPP systems (workload analysis), while other metrics are motivated by the core topics of this thesis, namely PE performance and the evaluation of branch prediction schemes within an NP system.

3.3.1 Prior Benchmarks and Analysis

The purpose of this workload analysis is to quantify the differing factors which determine NP and PE performance. Using a simulation model of the NP architecture while executing realistic workloads, the goal of workload analysis is to allow performance to be quantified. In the case of an NP architecture this may take the form of analysing the system bus utilisation as the number of connected PEs is increased. Previous work in the area of network processor workload analysis has tended to focus on defining network tasks via benchmark suites and searching for methods to utilise both thread-level and instruction-level parallelism within an architecture. In [90], [91] and [100], numerous network applications were defined, ranging from relatively simple tasks such as IP fragmentation to IPsec encryption. Memik et al. analysed nine applications for instruction mix on both the Intel IXP and a 4-way Superscalar processor [90]. Work by Wolf [100] and Byeong [91] analysed NP applications on a SPARC processor, investigating cache behaviour, instruction mix and Instruction Level Parallelism (ILP), but both were primarily focused on defining an NP-specific benchmark for future research. As was discussed previously, for reasons of cost, techniques such as data caching or ILP can be expensive to implement and may not be efficient on an NP. For example, the Intel IXP line of net-

work processors provide no cache mechanism and single in-order instruction execution. Ramaswamy presented an analysis of aspects such as memory access, unique instruction counts, data memory requests and per-packet instruction complexity across four header applications [159]. In [160] cache behaviour, instruction level parallelism and instruction sequences of the TCP/IP protocol stack were examined and compared to the SPEC benchmark, with a number of possible ISA extensions proposed, while a workload analysis of NP-based cryptographic algorithms was presented in [161], [162] and [163]. A limitation with the above scheme and analysis has been the lack of diversity in many cases. In most cases only a single application within a specific NP application group was used but, as was discussed in Chapter 2, in many cases there are a number of algorithms available to perform a specific function. In order to achieve an in-depth analysis of NP workloads it is the author's belief that multiple algorithms must be evaluated for each application group.

3.3.2 Branch Predictor Performance Evaluation

In Chapter 2 the concept of branch prediction was introduced. In general, this high level discussion utilised the predictor hit rate as the performance evaluation metric when comparing differing prediction architectures. Following on from this, a number of additional branch prediction metrics are defined in this section which will be used in later chapters to evaluate various branch prediction schemes within an NP system.

3.3.2.1 Branch Penalty Per Packet

Typically the branch penalty for a given pipeline depth is calculated as per Equation 2.5, where it is assumed that a taken branch evaluated in stage *M* requires *M-1* previous stages to be flushed from the pipeline. Since an NP will operate on a fixed data type, namely the packet, it is possible to calculate the total number of lost cycles due to taken branches on a per packet basis (Equation 3.4).

$$\tau_{pen} = (\rho_{tk} * N_{br} * P_{tk}) \tag{3.3}$$

$$\tau_{pen}(pp) = (\rho_{tk} * N_{br}(pp) * P_{tk}) \tag{3.4}$$

where ρ_{tk} is the ratio of taken branches to not-taken branches encountered during packet processing, $N_{br}(pp)$ is the total number of branches and $P_{tk}(pp)$ is the penalty associated with a single branch instruction. Where the majority of branches are unconditional or always taken (e.g. loop statements) the penalty will be small, while for highly conditional functions such as prefix tree traversal it can be assumed the branch penalty per packet will be highly dynamic from one packet to another.

3.3.2.2 Predictor Collision Rate

Along with the prediction hit rate, other metrics can be used to examine the performance of dynamic prediction schemes. As was previously discussed, dynamic predictors represent a type of hashing scheme in which the program address space is reduced in order to map branch instructions to a smaller and finite PHT structure. When implementing a hash-based addressing scheme, the trade-off which must be examined is how to reduce the cost associated with the hash table while ensuring a low collision rate between differing hash table entries. Unfortunately for branch prediction architectures, solutions such as complex (or near perfect) hashing algorithms, chained lists or bucket schemes are not applicable since the predictor logic must be at least as fast as the other stages within the processor pipeline and such schemes typically require a high degree of additional complexity. On the other hand, the effect of hash collisions (branch interference) can be minimised by increasing the PHT table size. Using this information it is possible to define the collision rate within a given predictor as the percentage of PHT entries which have at least two independent branches mapped to the same location.

3.3.2.3 Predictor PHT Utilisation Rate

Similar to the predictor collision rate, predictor performance can also be examined via the utilisation factor of the overall PHT. For an *n*-entry PHT, the utilisation factor is defined as the percentage of table entries which are used during execution. For general purpose processing, branch prediction schemes are designed with a constantly switching OS in

mind (in terms of the application being executed). A low table utilisation can be safely ignored since the next application will have a different footprint. Without either an OS or a constantly switching application, an NP-based branch predictor has an additional limitation since a hardware solution in which 75% of the pattern table is idle for long periods of time represents a significant waste of both chip area and energy.

3.3.3 PE Area Cost

In addition to these performance metrics, a fundamental evaluation metric when considering branch prediction is the area requirement of any solution. The small, highly optimised nature of NP applications has meant that complex hardware functions have not been needed (e.g. superscalar), while the parallelism inherent to network data flows has necessitated a parallel architecture. This PE parallelism, along with the low volume nature of the NP market, has made NP platforms highly sensitive to area considerations. When considering additional hardware such as branch prediction or caching, the fundamental question is whether the performance gain is justified when compared to the additional area required to implement this hardware. In order to accurately examine the design trade-offs when implementing a deeply pipelined architecture, an area estimate must first be derived for a typical PE design. Examining commercial architectures, the ARM7-TDMI and ARM9-TDMI architectures require approximately 75,000 and 110,000 transistors to implement without any cache [23]. Furthermore, the standard ARM architecture does not support floating point, similar to the configuration found within network applications where floating point functions are rare. When evaluating any RISC architecture employed as a network-based PE there are a number of additional components which may be added.

Firstly, the program memory must be implemented on-chip in order to minimise the instruction latency, with the program memory coupled to each PE (See Section 4.2.2). In addition to the cost associated with the on-chip control store, two additional area requirements must be factored in when calculating the area cost of a typical PE architecture. The need to hide long access latencies to external devices necessitates some form of hardware based multi-threading, allowing greater PE utilisation to be extracted. Finally, the use

of multiple processing engines, each supporting multiple threads, requires some form of buffering when accessing bussed external devices. Whereas a single PE system could simply latch memory access in control logic connected to the system bus, a multi-PE, multi-threaded system must buffer multiple commands in a FIFO structure connected to the system bus (or buses). Using the area of the ARM9 microprocessor as a base (110,000 transistors), the area of a PE architecture can be estimated as follows. Firstly, an *n*-word by 32-bit program memory would require $n * 32 * 6$ transistors. The area for the hardware based multi-threading is negligible except for the need to parallelise the register bank. Assuming each of the *m* threads has its own 16-register bank, the area of the register bank is increased by a factor of *m*. Other costs associated with the multi-threading can be ignored since it can be implemented via a simple round robin scheme. Finally, with each of the *m* threads accessing the shared bus, an *m* entry input and output FIFO are required to buffer data transfers between the PE and other external devices. For an *l* bit bus, each FIFO would require $l * m * 16$ bits to be stored.

Using the sample configuration outlined in Table 3.1, the program memory would require 393,000 transistors, while the register base and I/O FIFOs would require 40,900 (*16 * $N_{threads}$ * 32 * 10*) transistors and 40,900 (*160 * $N_{threads}$ * 16*) transistors respectively. Since the area of the program memory is much larger than all other components within the PE design, the cost associated with it is not included in any comparison. The area of a similar ARM9-based architecture is therefore increased to $110,000 + 40,900 + 40,900$ transistors. Future branch prediction trade-offs are examined with respect to this area estimation.

Table 3.1: PE System Parameter

Section	**Name**	**Parameter**
Program Memory	P_{size}	2K Words
Thread Count	$N_{threads}$	8
Bus Data Width	W_{data}	128
Bus Cmd Width	W_{cmd}	32

3.3.4 Area Cost of Branch Prediction

Recalling the area estimates given for the various 2 level predictors outlined in section 2.7.2 it is clear that for a global address scheme, such as *GAg* or *gshare*, any expansion of PHT increases the area cost. For the per-address schemes, increases in either the PHT size or the number of GHR entries will dramatically increase the number of transistors required. As an example, consider the three prediction architectures examined in [90]. In the first case a 2KB (8K entry) global history scheme is examined. Ignoring the cost associated with the address logic, the area of such a predictor can be estimated at:

$$\begin{aligned} A_{GAg}(2KB) &= (6 * 2 * 8192) + (16 * log2(8192)) \\ &\approx 98,500 \end{aligned}$$

With a maximum PE area (excluding program memory) estimated at 200,000 transistors it is clear that predictors of this size would not be justified on an NP platform. Assuming a 16 PE system, 16 predictors matching the above configuration would require the same amount of transistors as 8 additional PEs. The two other architectures examined in [90] were a 2KB-4KB bimodal system and a combinational predictor involving both the 2KB GAg predictor and the 2KB-4KB bimodal system. In the case of the bimodal system, the 4KB tag cache would require almost 200,000 transistors for the SRAM alone, with a large amount of transistors required for the lookup logic (equivalent to Content Addressable Memory (CAM) logic) and the first level prediction tables. Without large increases in the area complexity of a PE (additional program memory, level 1 cache, superscalar) such large predictor structures are not justified.

3.4 Conclusions

This chapter presented an overview of the difficulties regarding performance evaluation of NP architectures. With modern SoC-based NPs comprising multiple hardware modules, parallel processors and a complex memory hierarchy, a major challenge to NP researchers

is how such an NP architecture can be efficiently and accurately modelled without requiring a specific model for each architecture and configuration. Examining the current state of the art within the domain of computer modelling and simulation it was found that while purely mathematical models allow rapid development it is difficult to estimate some of the parameters needed to make such a model accurate. Other modelling techniques can be generally described as simulators which attempt to mimic the hardware target. Within the domain of digital simulation there are a number of abstraction layers possible based on the degree of precision needed. Whereas complex cycle-accurate models allow highly accurate simulation, a higher level functional simulator will tend to favour speed (of development) and flexibility, sacrificing some of the accuracy achievable with a cycle-accurate model. Unfortunately there is no unified NP simulation model currently available to NP researchers which allow architecture aspects such as PE, memory and bus configurations to be examined. In the next Chapter a new NP specific simulator is proposed which attempts to solve this problem, allowing fast NP simulations while at the same time providing the flexibility to allow different NP architectures to be explored.

CHAPTER 4

A New Simulator for Network Processors

4.1 Overview

In Chapter 2 an overview of the current state of the art within NP design was presented, outlining the common components which comprise a modern NP while also highlighting the different NP architectural configurations currently available. In Chapter 3 it was argued that the lack of a single unified NP architecture made research within the NP domain difficult. When analysing a computer system to evaluate design aspects such as memory usage, branch behaviour or instruction distribution a simulation model reflects the most powerful method of evaluating such a system. With this in mind, in this Chapter a new NP-based simulator is proposed which attempts to model the configuration blocks commonly found within a modern NP while providing enough abstraction such that simulation models can be rapidly and efficiently developed. Written entirely in C and using the widely used ARM architecture, this simulator, called SimNP, follows a part-functional model, with simulated sub-modules implemented as cycle-accurate (e.g. memory latency) but where the interconnection system remains at an abstraction layer high enough to allow differing NP architectures to be explored.

4.2 SimNP Simulator

Summarising the performance evaluation survey presented in the previous chapter it is clear that within the topic of system simulation a number of trade-offs are apparent. While stochastic models allow rapid evaluation of system performance by abstracting away architecturally defined parameters, the assumptions regarding traffic distribution, inter-process communication and queue capacity can introduce significant errors. On the other hand, a software-based simulation framework provides an alternative means of examining system performance. Modelled at either a system level or by interconnecting various subsystem components, a simulation framework provides a mechanism of improving the precision of salient information by trading off some of the high level abstraction capable with a mathematical model. In this section a brief outline of the proposed SimNP simulator is presented. Designed to incorporate the components common to a modern NP, it allows rapid analysis of NP systems. Figure 4.1 shows the system block diagram of the SimNP simulator, highlighting the default components simulated within the model. An overview of the software architecture employed by SimNP is also presented.

4.2.1 Software Architecture

The software architecture employed within SimNP follows the outline shown in Figure 4.2 [1]. Following an execution-driven method, applications are created in C or C++ before being compiled into static binaries for execution using an ARM targeted cross-compiler (The work presented in this thesis used the open source *gcc* compiler suite [103]). A number of packet traces are supported, with SimNP attempting to rebuild valid packet traces from some of the popular anonymised trace files available via the NLANR repository [164]. Since these packet traces remove any sensitive data, new IP addresses are derived from the MAE-west and AT&T East Canada routing tables. Once the header has been rebuilt, random data replaces the packet payload. To the best of the author's knowledge, there is no current methodology or framework which allows packet payload information to be rebuilt to a realistic level. At simulation runtime, each simulation component is

[1] The shaded areas denote modules implemented by the author.

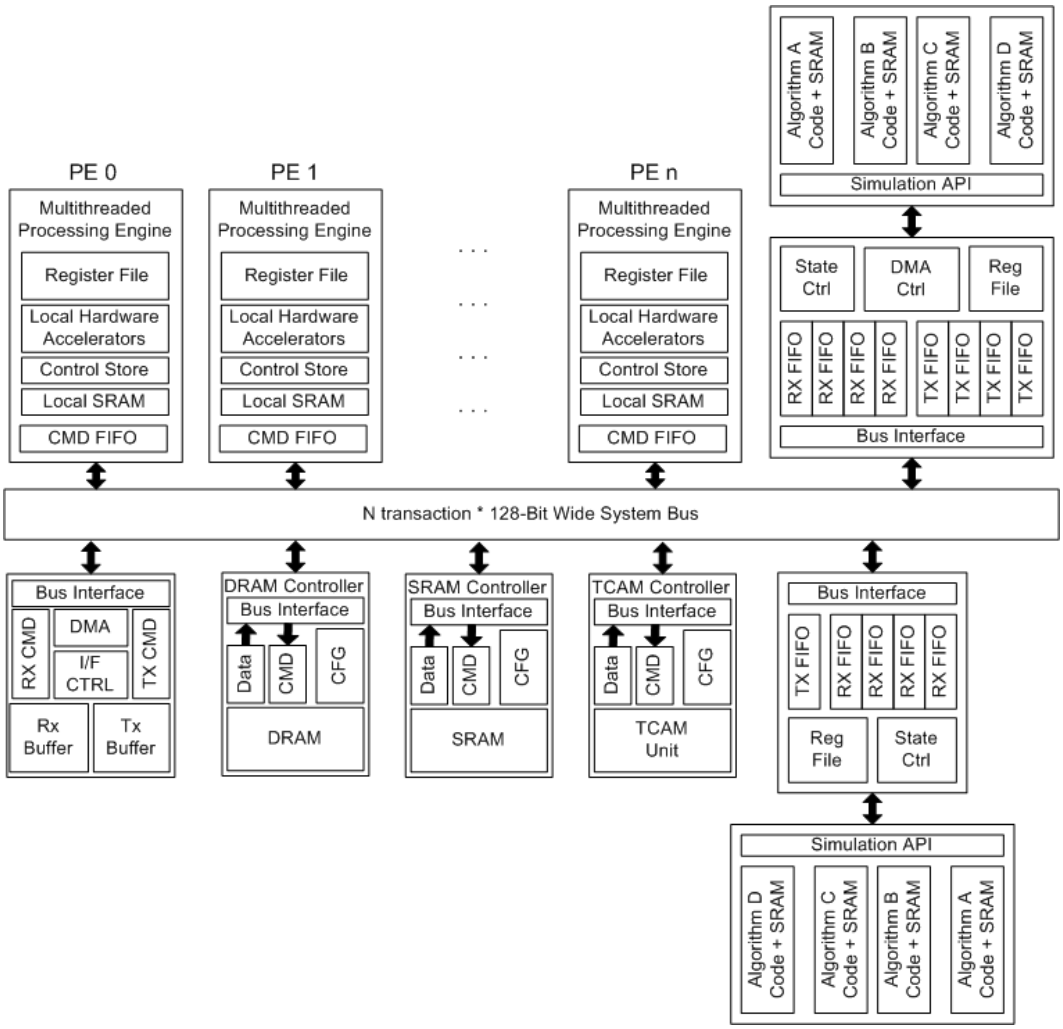


Figure 4.1: SimNP Block Diagram

initialised within its own space, interacting via the central bus and simulator core. At initialisation, the simulation configuration file defines the simulator parameters. Some of the parameters, which can be configured at run-time, are outlined in Table 4.1. For all devices it is also possible to configure the clock frequency as well as the FIFO depths. Target applications to be simulated can be written in either assembly, C or C++, with the applications following a run-to-completion model. Typically, a PE can request a packet from a central arbitrator (either the interface unit or another PE configured to maintain the packet queues). During execution, operations involving long latencies will trigger the current thread to be placed in a waiting state while the operation is being completed, allowing another thread access to the PE.

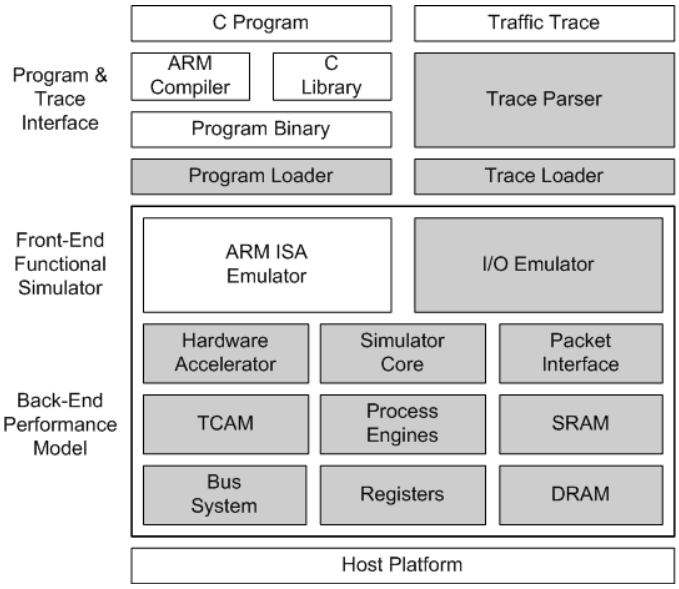


Figure 4.2: SimNP Software Architecture

4.2.2 SimNP Processing Engines

In general a PE can be defined as a Harvard-based, cache-less, integer-only RISC microprocessor. While the lack of a data caching can be justified by the low spatial data locality of NP applications, the justification for employing a Harvard architecture can be seen when common NP applications are examined. Firstly, the amount of processing which can be applied to any packet traversing the network is limited by the upper bound determined by the need to maintain wirespeed. When compared to large programmes such as OS kernels or web browsers, the number of instructions within an NP system is much smaller. In Table 4.2 the binary code size (in KB) is given for common NP applications when compiled for the ARM architecture (without optimisation). The second characteristic is the relatively static nature of NP applications. Unlike a general purpose system where the executing application may change many times every second, the same NP application may remain in place for long periods of time with only minor changes or alterations to program flow. For an NP system it is therefore possible to provide a small amount of closely coupled on-chip SRAM, where the instruction data can be fetched without the need to access external memory.

The other common trait within PE design is the provision of hardware based multi-threading. The primary advantage of multi-threading is as a mechanism for hiding latency

Table 4.1: SimNP Configurable Parameters

Application	Parameter
PE	Number of PEs
	Number of Threads
	Local Data/Control Store
Bus	Latency
	Bandwidth
Memory	Access Time
	Data Width
Hardware Acceleration	Algorithm
	Device Latency
	Process Delay
Interface	Buffer Size
	Bus Width

when accessing slower external devices. Largely transparent to the programmer, multiple threads in hardware can be achieved through a switch of the register bank. More complex schemes involve implementing the entire 'pipeline' on a single PE, with the context switch providing a means of switching between stages of a pipeline. For SimNP, the multi-threading is designed to follow either an automatic or manual trigger point. During automatic thread switching, a long latency operation causes a PE to switch thread to the next available thread. Once the request has been processed, the PE is released into the PE waiting pool. The manual configuration follows a similar design flow with the exception that the context switch is triggered by a segment of volatile assembler code.

Table 4.2: NP Application Code Size

Application	Code Size (KB)	Application	Code Size (KB)
AES	1914	TRIE	634
CAST	2182	HASH	630
RC4	1670	HYPER	720
SHA1	3248	RFC	438
MD5	2375	TCM	380
FRAG	382	TBM	1264
CRC	359	DRR	685
RS	3400	STAT	829

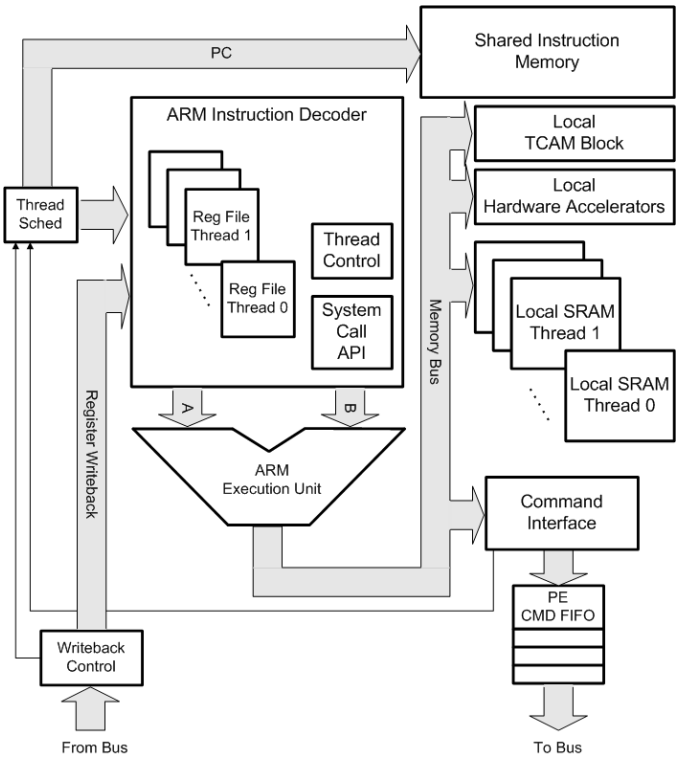


Figure 4.3: SimNP Process Engine

With regards to the ISA employed by SimNP, the ARM architecture was selected for the SimNP platform since it allows the SimpleScalar execution unit to be reused [127], while also allowing the user to take advantage of mature development platforms.

Two additional modifications are also incorporated in the SimNP PEs. Firstly the ARM software interrupt instruction (*swi*), which allows hosted applications to access system call functions such as file *read* and *write*, can be reused to simulate new instructions since high level system calls are not applicable to NPs. The (*swi*) instruction can therefore be used to rapidly map new instructions to the system without needing to change the compiler specification. The second addition is the provision of local hardware accelerators which allow low access hardware blocks to be simulated. Like the shared hardware blocks outlined in Section 4.2.6, the local hardware blocks allow functions to be benchmarked under conditions where only inter-thread contention determines access latency. The system block model for the PE employed by SimNP is shown in Figure 4.3.

4.2.3 SimNP Memory Hierarchy

The advantages of one memory hierarchy over another are largely determined by the choice of microprocessor architecture. Modern general purpose architectures typically use a flat memory model with devices mapped to specific regions within the memory space. When compared to a more complex system involving specialised instructions for accessing specific external blocks, the flat model presents a clean, cheap and efficient memory hierarchy. At a technology level, there are a number of traits common to NP platforms. In general, the consensus has been to store packets in slow DRAM technologies while more latency sensitive control data is stored in external SRAM. More cost effective than a full SRAM solution, this hybrid model is common to commercial architectures. With the SimNP PEs employing a flat 32-bit memory space, the entire SimNP memory model is simplified to a linear memory space from between addresses *0x00000000 - 0xFFFFFFFF*. The majority of the memory is undefined and can be used to map new functions. A small section of the memory is used as the base address for DRAM, SRAM or TCAM devices connected to the system bus. Each memory device can be altered in either size, location or even whether the device is accessible by all devices in the NP architecture. A small number of memory locations are reserved for memory mapped precompiled library functions (Interface Unit Access, *printf*). A sample SimNP memory model is shown in Figure 4.4.

4.2.4 SimNP Inter-Device Communication

For a functional simulator, it is possible to model the system bus using a small number of factors; the arbitration method, the cycle time and the bandwidth. For a transaction-based bus which moves N_{cmd} commands, each of which is M_{width} wide, the bandwidth is:

$$B_{width} = N_{cmd} * M_{width} * clk_{bus} \tag{4.1}$$

Within the SimNP platform, the device bus is implemented as a fixed-length command driven bus with central bus arbitration. Similar to the AMBA bus, the SimNP bus aims to

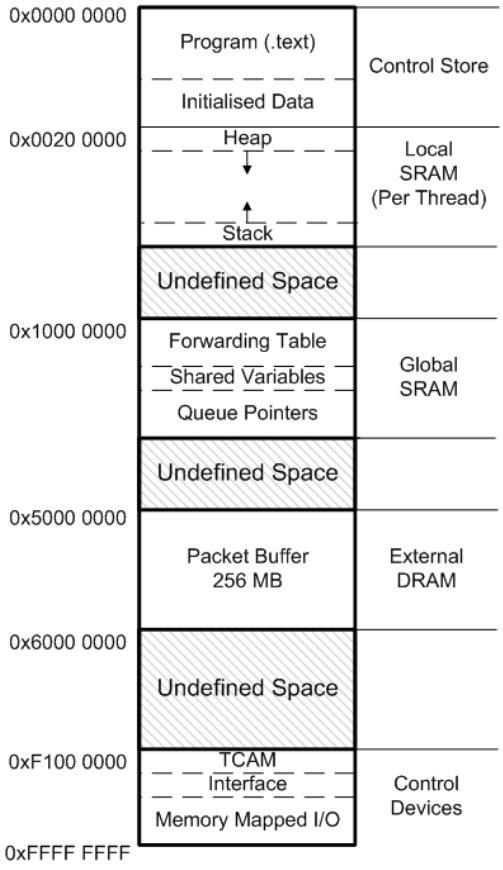


Figure 4.4: SimNP Sample Memory Map

remove the underlying bus specifications from the simulation model. Devices are assumed to contain either one or more receive and transmit queues. On each bus cycle the bus unit is configured to route N_{cmd} waiting commands from one source TX First-In-First-Out (FIFO) to another addressable RX FIFO. For the current cycle the bus attempts to route a single transaction from the device which should next have access to the bus. In the event of no command being found at the device, TX FIFO, the bus unit checks the next available device until all nodes have been processed. In this manner the bus unit operates as an optimised implementation of the round robin algorithm and ensures high bus utilisation.

4.2.5 SimNP Interface Unit

On an NP the interface unit allows the packets to be transferred in and out of the NP, either to the network or the switching fabric within the routing architecture. Interface standards, such as GMII, SPI-4.2 and CSIX, are commonly found on NP architectures and allow an NP to connect to physical connections (optical, switch fabric or Ethernet). Within a func-

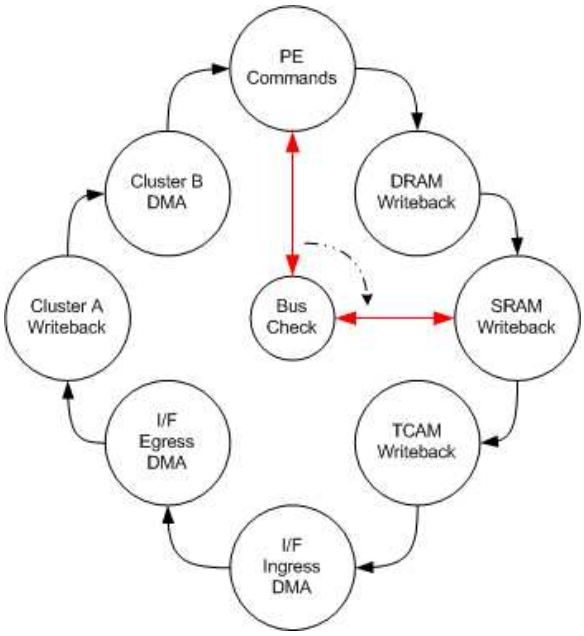


Figure 4.5: SimNP Bus Model

tional framework, it is possible to generate an abstract model for these communication models. Figure 4.6 presents a block diagram for the CSIX interface which allows devices such as traffic managers and NPs to connect to the switching fabric. As can be seen from the diagram, only a small number of signals are required, i.e. the data bus, along with a clock, parity and start of frame control bit. The transmit unit from the switching fabric to the egress NP follows a similar outline. The CSIX specification [165] defines the data bus as 32, 64, 96 or 128-bits while the clock signal is defined in the range of 100-160MHz.

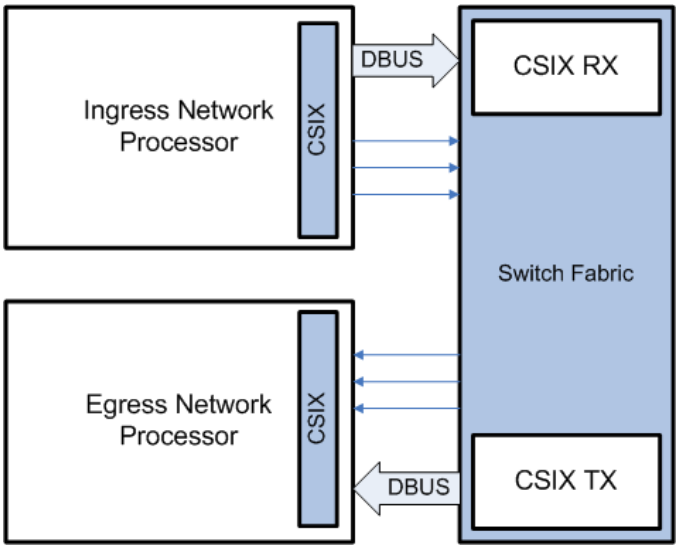


Figure 4.6: Common Switch Interface [165]

A similar outline is defined for the SPI Interface, while GMII-type connections employ an 8-bit Ethernet framing method. The SimNP interface is designed to model the behaviour of an SPI-type interface without requiring details of the underlying hardware or specifications. Incoming packets are transferred to the receive FIFO which is maintained in SRAM coupled directly to the interface unit. Both ingress and egress packet buffers are maintained in round robin fashion, with packets automatically dropped once the buffer fills. It is assumed that any realistic simulation would attempt to meter the traffic after initial buffering, using either a second stage hardware block or a PE configured to manage packet flows. When requesting packets from the interface, a PE will issue a request command containing the address where the head packet is to be moved to. A similar process is followed when adding a processed packet to the TX buffer queue. Transfers, either to or from the interface unit, are handled via a DMA controller which can use either the main system bus or a dedicated bus to transfer data through memory. Two bus methods are provided as the volume of command requests generated by the interface can be very large for small parallel architectures, a condition which can result in high bus contention rates.

4.2.6 SimNP Hardware Acceleration

Simulation of hardware acceleration involves deriving a framework for classifying hardware accelerators based on the functionality provided by the hardware accelerator. For parallel systems such as an NP, hardware acceleration can be provided in three different methods.

- PE Independent Acceleration - Acceleration blocks which operate largely independent of the application running on the PE array. For example, consider an NP system which includes an on-chip mechanism for congestion control and queue distribution of incoming traffic. Within the SimNP platform, PE independent accelerators such as ingress metering or queue maintenance can be inserted to intercept incoming packets automatically and provide the required functionality.

- PE Shared Accelerators - These hardware blocks are assumed to be shared across all PEs and are accessed via the PE memory map. Since these devices are shared it can be assumed that contention will account for a significant amount of processing delay. Shared hardware blocks are commonly those applications which would be too expensive (in area cost) on a per-PE basis, for example packet encryption and decryption.

- Per PE Accelerators - Typically much more simple hardware blocks, per-PE hardware accelerators typically provide functions which are commonly used by all PEs and NP applications. For example, hash key generation is often used within NP applications to index packet control information in a hash structure. Since all PEs would have to access this structure it can significantly improve performance if the hash index generation is offloaded to hardware coupled directly to each PE.

Within any simulation model, the salient information regarding these various hardware blocks is; the queue delays associated with accessing the hardware block, the device latency and whether the underlying architecture is pipelined or non pipelined. Using these parameters, it is possible to implement a functional model of the hardware block without sacrificing any precision within the results. For example, a shared hardware block providing deterministic processing (e.g. packet classification rule lookup) would require T_n cycles to process a single packet.

$$T_n = t_{access} + t_{process} + t_{result} \tag{4.2}$$

where t_{access} is the access time associated with a shared device, largely comprised of the contention between PEs and the bus (and or) hardware device. The processing time for the hardware block is given by $t_{process}$ and is determined by the underlying architecture. If the hardware block is assumed to be pipelined, processing multiple commands on each cycle, the processing time can be calculated as:

$$t_{process} = \frac{s + (p-1)}{f_{hw}} * f_{pe} \tag{4.3}$$

where s is the number of pipeline stages, p is the number of stages occupied by one command, f_{hw} is the clock frequency of the hardware and f_{pe} is the clock frequency of the PE issuing the command. Within SimNP, two application interfaces are provided which allow memory mapped hardware blocks to be modelled. Each interface supports four independent accelerators, allowing multiple server configurations to be examined. In the case of the upper hardware interface, cluster A, the DMA and state controllers target payload applications and can be configured by writing the *source data address*, *destination data address* and *data length* to the hardware control registers. The lower hardware interface allows header based functions such as forwarding and classification to be simulated. Since both interfaces are memory mapped, it is a trivial task to reconfigure the hardware blocks if additional parameters must be passed to the hardware block.

4.3 Comparison with Existing Solutions

The design goal of SimNP is to provide a platform for the study of network processing systems. Consequently, it is a simulation infrastructure with a collection of commonly used architectural features rather than a model of any existing NP system or platform. Using six common network applications (outlined in Chapter 5), a comparison between the SimNP and the SimpleScalar platforms is presented in the following sections. Summarising, two of the applications (*AES* and *MD4*) represent block based cryptographic algorithms used for functions such as IPsec encryption and authentication. Both algorithms require a large degree of processing, involving extensive logic and arithmetic operations, and allow raw processing metrics such as Million Instruction Per Second (MIPS) to be determined. A third payload application, *FRAG*, involves requesting a packet for processing, checking to see if the packet is large enough to be fragmented, before dividing the packet into smaller sizes. Unlike the other payload applications (AES and MD4), the major difference of the *FRAG* algorithm is the large amount of memory operations required but with very little processing applied to each packet. In addition to these three payload applications, three header processing functions are also examined. The *TRIE* application performs IP packet forwarding, utilising the AT&T East Canada routing table. Each

packet is verified before the next hop is generated and the packet is queued for egress transmission. The second application, *STAT*, uses a hash-based structure to maintain information on a per-flow basis. New flows are allocated space within the structure while terminating packet flows are discarded after any per-flow statistics are added to the global statistics database. The final application examined is the Deficit Round Robin (*DRR*) algorithm which provides a mechanism for balancing variable length packet queues. Relatively simple to implement, the *DRR* algorithm only moves a pointer to the packet around memory, since memory copy routines can be computationally expensive on load/store architectures such as the ARM platform. Each application is benchmarked using two packet traces obtained from the NLANR repository. The PSC trace is gathered from an OC-48 connection while the AMP trace captures datagrams traversing the OC-12 connection.

4.3.1 Simulation Time

Within the computer architecture domain, research aspects such as simulation time have become less important due to the performance and cost developments within general purpose systems. For research purposes any simulation platform must remain fast enough to allow rapid analysis for configurations where longer term analysis is required. For NP systems, this second point becomes more important when trends within network traffic are factored in. While analysis such as instruction distribution and memory analysis can be performed using small trace files, metrics such as power analysis, load and queue balancing, etc. may require network traces running for long periods of time, capturing millions of packets. For an application such as *AES* or *MD5*, which require extensive computational resources, a simulation involving 100,000 packets takes approximately 45 minutes to run on a standard 2GHz Intel Core Duo laptop. Executing significantly fewer instructions, header applications such as such as *TRIE* require only 31 seconds to process 100,000 packets. For both header and payload applications the simulation time is linear, allowing an accurate simulation time to be extrapolated from smaller simulation runs.

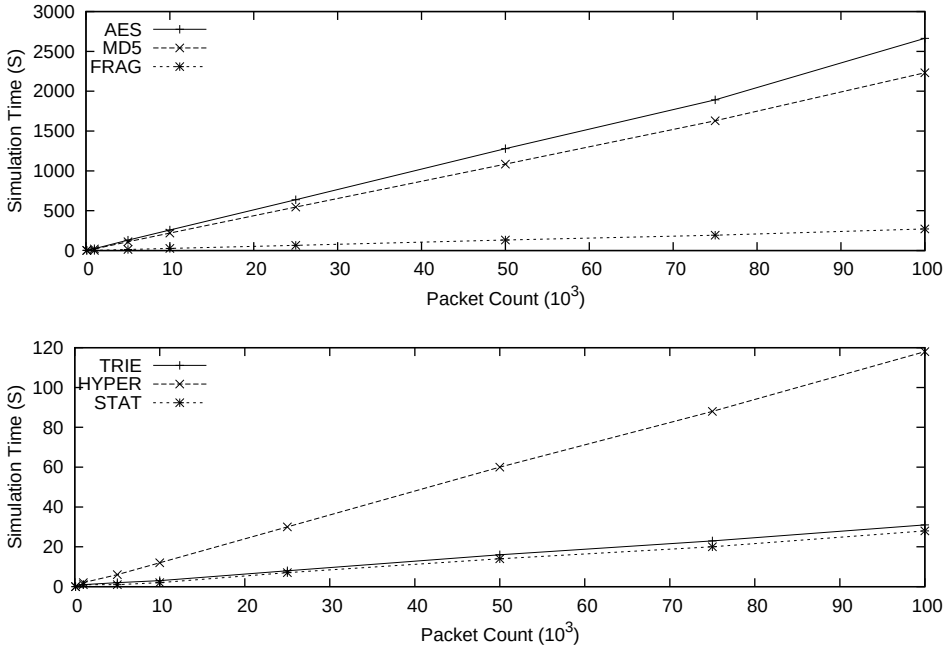


Figure 4.7: SimNP Simulation Time

4.3.2 Simulation Performance

In addition to simulation time, a number of other metrics can be used to verify performance. For the payload functions, the number of instructions simulated per second allows a method of comparing various simulators and configurations. In addition to the simulated MIPS rate, NP performance can be quantified using either the number of processed Packets Per Second (PPS) or the number of bits processed per second (Kilo-bits per second). In the case of payload applications, the number of bits processed per second is a more useful metric since the processing cost per packet is determined by the packet length. On the other hand, the number of packets processed per second provides a reasonable metric when comparing header based applications.

Examining the results in Table 4.3, the average simulated MIPS rate is 2.39, approximately 30% higher than the equivalent SimpleScalar/Packetbench configuration. Similarly, the average number of bits processed per second (Kbps) with SimNP is approximately 50% higher for the AES and MD5 applications when compared to the previous solution. For the header applications it can be seen that both the simulated MIPS and bit processing rate can be heavily skewed by the underlying traffic. Since header applications

Table 4.3: Performance Analysis of SimNP Vs. Simplescalar/Packetbench

Application	Trace	SimNP			Simplescalar		
		MIPS	Kbps	PPS	MIPS	Kbps	PPS
AES	AMP	2.23	25.98	29.99	1.28	17.88	21.37
	PSC	2.27	26.51	36.34	1.29	18.52	24.51
MD5	AMP	2.65	36.24	42.34	1.12	20.78	24.27
	PSC	2.47	31.82	44.25	1.10	19.59	27.25
FRAG	AMP	2.35	357.83	364.96	3.28	1680.13	2000
	PSC	2.35	359.16	438.60	2.90	1406.03	2000
TRIE	AMP	1.50	1866.81	2222.22	5.03	60.44	71.94
	PSC	1.57	1634.92	2325.58	5.18	52.08	74.07
STAT	AMP	1.65	2470.78	2941.18	3.27	1400.11	2500
	PSC	1.52	1900.04	2702.70	3.92	1406.03	3000
DRR	AMP	2.01	4000.31	4761.91	3.89	2800.22	3333.33
	PSC	1.93	3195.52	4545.46	3.47	2343.38	3333.33

typically involve only a small amount of processing per packet, performance for SimNP is limited by two factors. Firstly, when performing workload analysis, SimNP 'block' fetches enough packets to fill the ingress buffer, incurring a small penalty since excess packets may be buffered. Secondly, SimpleScalar simulated all memory operations as atomic, with a latency of only one cycle. Since SimNP connects multiple devices via a shared bus system it is unlikely that a memory operation would be arbitrated in a single cycle, even if all devices are running at the same clock frequency.

4.3.3 Workload Validation

Along with performance evaluation, a brief workload analysis is presented in order to validate SimNP when compared to a SimpleScalar/Packetbench configuration. Using an OC-3 packet trace, the six applications are compared in Table 4.4. Examining the data, a number of reasons can be deduced to explain the performance difference. Firstly, SimpleScalar requires the use of a more complex standard c library (*glibc*) since packet request functions are complied as file read *fread* and file write *fwrite* functions. The library also allows more complex instructions such as the ARM load and store multiple (LD-

M/STM) to be invoked, instructions which are not assumed with the lightweight *newlib* library. In addition to the instructions used to fetch and store data between memory, the difference in memory partitioning accounts for some of the variance. SimpleScalar assumes a flat memory space, with each packet modified 'in position', without any need to copy the packet to a local space. A SimNP model will typically copy a packet from the global packet queue then modify the local copy before returning the processed packet, while the SimpleScalar simulator assumes the packet does not need to be copied before processing. As a final note, SimpleScalar requires the standard library (*glibc*) to be used for functions such as I/O, while the SimNP platform can use either *glibc* or the more lightweight *newlib* library.

Table 4.4: Instruction Distribution for SimNP & SimpleScalar

Application	Simulator	Load	Store	U-Branch	C-Branch	Logic
AES	SimpleScalar	30.66	6.94	1.48	3.21	57.71
	SimNP	38.49	16.84	1.03	2.44	41.21
MD5	SimpleScalar	40.79	13.54	3.91	3.85	37.91
	SimNP	34.36	12.56	1.45	1.54	50.09
FRAG	SimpleScalar	22.43	20.35	2.77	6.67	47.79
	SimNP	21.76	20.57	2.29	12.76	42.62
TRIE	SimpleScalar	16.07	12.17	2.58	11.82	57.36
	SimNP	28.38	16.7	4.54	6.75	43.63
STAT	SimpleScalar	32.51	20.02	1.58	8.43	37.32
	SimNP	27.68	16.11	3.53	9.65	43.04
DRR	SimpleScalar	32.33	17.35	1.05	11.98	37.28
	SimNP	26.53	11.91	1.65	5.62	54.3

4.4 Conclusions

This chapter has described a simulation framework for NP architectures. While a number of techniques were evaluated, it was decided that a simulator could be developed which would provide good accuracy in terms of processing speed and latency, while at the same time allowing architectural aspects such as the number of PEs, memory hierarchy and

hardware acceleration to be thoroughly evaluated. Employing a modular framework similar to a System-C transaction level simulator but implemented within a single high level language, the simulator proposed does not require long simulation times and is capable of being run on a general purpose desktop computer or laptop. Modelling the components common to a packet processing environment, SimNP currently includes simulation models for multi-threaded process engines, external DRAM and SRAM, network interfaces, TCAM devices and hardware acceleration blocks. Supporting the ARM ISA it allows programmes to be implemented in high level languages such as C or C++, removing the development bottleneck associated with mapping algorithms and applications to assembly code. The use of a memory mapped I/O allows rapid addition or removal of architectural features, as well as complex network processor design space exploration, balancing a flexible and appropriate abstraction level while providing meaningful statistics and analysis.

Benchmarked to previous solutions it was found to provide similar performance across a wide range of applications, while allowing greater accuracy to be extracted from simulations. In the next chapter, this simulation framework is used to examine NP performance workloads, as well as to examine architectural techniques such as parallelisation or hardware offloading.

CHAPTER 5

Analysis of NP Workloads

5.1 Introduction

As was previously discussed, the tasks implemented on a modern router have expanded from basic packet switching to more computationally intensive applications such as security, classification or payload modification. As such, it is important to develop an understanding of NP applications via a workload analysis.

Using the information obtained from a workload analysis it is possible to guide future development and optimisations of the architecture being analysed. In the case of an NP platform, a workload analysis allows potential bottlenecks to be identified, while common penalties such as bus contention or branch penalties can be quantified. Unlike an application benchmark, which attempts to define applications to be used as a means of comparing one architecture and another, a workload analysis uses a broad variety of NP applications and algorithms. With numerous applications capable of performing the same function on an NP platform, selecting multiple applications from each group ensures the analysis remains valid for various NP configurations.

For a pipelined PE, the processor utilisation is maximised by ensuring all stages of the pipeline are processing data at all times. While the primary focus of this thesis is the effect of conditional branches within the pipeline, there are a number of additional aspects which directly affect PE utilisation. These include the memory behaviour and distribution

of NP applications, the bus utilisation of a multi-PE system and the conditional behaviour of NP applications.

Developing both an analytical framework as well as a quantitative evaluation of NP workloads and systems, the work presented in this chapter provides a detailed understanding and examination of NP workloads. In all cases the simulation models used were designed to reflect realistic operating conditions. Using the SimNP simulator model outlined in the previous chapter, a large suite of NP applications and packet traces are used to fully explore the architectural aspects which affect NP performance.

5.1.1 Network Processing Complexity

In commercial NP architectures it is possible to define two very broad architectural philosophies for use in PE design. In the case of Xelerated, Bay Microsystems and Ez-chip, it is assumed that NP applications can be implemented in a fully deterministic fashion, with a given processing rate guaranteed. Requiring a pipeline design, an example of an NP which maximises deterministic behaviour can seen in the architecture employed by Xelerated, which is constructed around a pipeline of special process engines with each engine capable of executing up to 4 instructions per packet. On the other hand, architectures employing a more traditional RISC structure such as Intel (Netronome) and Cavium leave timing issues to the customer, allowing maximum flexibility in terms of how the NP resources are deployed.

At first inspection it would appear that deterministic processing is well suited to NP systems. Consider an NP which is processing incoming packets arriving at the NP ingress port at the rate of *LR* Megabits per second. To maintain the line rate of the incoming traffic, the NP must therefore ensure that the amount of time taken to process any packet does not exceed the packet inter-arrival time $T_{process} \leq T_{IR}$, where T_{IR} is the packet inter-arrival time between two minimum sized packets, each P_{size} bytes (Equation 5.1).

$$T_{IR} = \frac{P_{size} * 8}{LR} \tag{5.1}$$

Table 5.1: Optical Carrier Instruction Budget

Interface	**P**$_{size}$	**Line Rate (Mbps)**	**T**$_{IR}$**(nS)**
POS OC-3	46[34]	155.52	2366
POS OC-12	46	622.08	592
POS OC-48	46	2488.32	148
POS OC-192	46	9953.28	37
GigE	64	1000	512
10GigE	64	9953.28	51

In Table 5.1 the minimum packet inter-arrival delay for a number of network configurations is shown. As can be seen from the table, each generation of network technology has the effect of reducing the available number of clock cycles by a factor of 4. A similar calculation can be done for other configurations, for example ATM-over-SONET utilises fixed 52-Byte frames sent over an optical SONET network.

5.1.1.1 Deterministic Processing

Within *P*-stage deterministic NPs, the entire application is divided across these stages at compile time. Since not all pipeline stages will have access to the same resources, the application must be divided with this restriction in mind. For example, access to an external search structure might be reserved to a single stage of the pipeline. The compilation algorithm must therefore attempt to divide the application into *P* even stages while also ensuring this hardware limitation is accounted for. There are a number of challenges within such pipelined NP architectures. Firstly, not all applications are suited to pipelining, with certain functions requiring to be atomic operations. Secondly, how can those applications which are well suited to pipelining be efficiently partitioned when hardware limitations are factored in? And thirdly, how can deterministic processing be ensured when technologies such as DRAM require a variable number of cycles to complete?

By implementing certain dedicated hardware solutions it is possible to solve some of the contention and latency issues with a software based processing system. The most common NP-based example of this involves implementing on-chip logic to calculate the next hop address to which a packet should be forwarded to. Instead of utilising a software

traversed trie structure, requiring multiple memory accesses, a hardware accelerator capable of traversing the trie in a single cycle can be used to place a known upper bound on the processing time of each packet, with queue models such as those outlined in Chapter 4 capable of modelling such a system as a basic Markovian queueing system.

5.1.1.2 Exponential Processing

While a deterministic NP architecture places an upper bound on the application which can be supported, a PE design which is fully software based allows the application designer to determine the tradeoff between the degree of application complexity which can be supported (the number of cycles available for processing) and processing rate which must be maintained in order to meet the target network line rate. In the case of a non-deterministic application, the processing time is discrete in nature within two bounds. Consider a RISC-type PE executing application *n* on packet *p*. The processing time $T_n(p)$ is given by:

$$T_n(p) = \sum_k t_k \tag{5.2}$$

where t_k is the time delay associated with function *k*. In general, the majority of the processing time is due to the instructions executed and the latency associated with IO and memory operations:

$$T_n(p) = t_{insn} + t_{mem} \tag{5.3}$$

With only small changes in the underlying data structure, the number of possible execution paths is finite and can be estimated at compilation. To illustrate this, consider the code segment outlined in Listing 5.1. The function outlined represents an IPv4 address verification routine and is required during IPv4 packet forwarding. Given a 32-bit IP address, a router must check if the IP address is either invalid, multicast range or a *normal* routable address. In the event of an invalid address the routing application must be signalled to drop the packet, while valid packet addresses, either multicast or unicast, must be processed in their respective manner. Viewing each conditional operation as a node, it is possible to view the application as a path execution tree with a branch at each

node. In Figure 5.1 the execution path for the code segment is shown. In all there are four bitwise checks which are mapped to three possible return nodes. In terms of possible paths traversed, it can be seen that there are five possible combinations.

```
void check_address(unsigned int ip_address) {
unsigned int vaddr;
    /* check for 0.x.x.x and 127.x.x.x */
        vaddr = (ip_address >> 24) & 0x0000000F;
        if ((vaddr == 0) || (vaddr == 0x7F))     /* A */
                return INVALID_TYPE;
        /* check for 255.255.255.255 */
        if ( vaddr == 0xFFFFFFFF )               /* B */
                return VALID_MULTICAST;
        vaddr = (ip_address >> 28) & 0x0000000F;
   /*
    * Check for 224.x.x.x to 239.x.x.x(multicast)
    */
    if(vaddr == 0xE)                             /* C */
                return VALID_MULTICAST;
    if(vaddr == 0xF)                             /* D */
        return INVALID_TYPE;

    return VALID_IP;
}
```

Listing 5.1: Verify IP Address

```
A-INVALID        A-B-MULTICAST      A-B-C-MULTICAST
        A-B-C-D-VALID        A-B-C-D-INVALID
```

While t_{insn} is discrete, the time delay associated with memory operations is determined by a number of factors. Memory access can be subdivided into access delays associated with three regions, t_{pkt} which covers memory accesses to packet memory, t_{ctrl} which are those memory operations accessing the control memory and t_{local} which defines

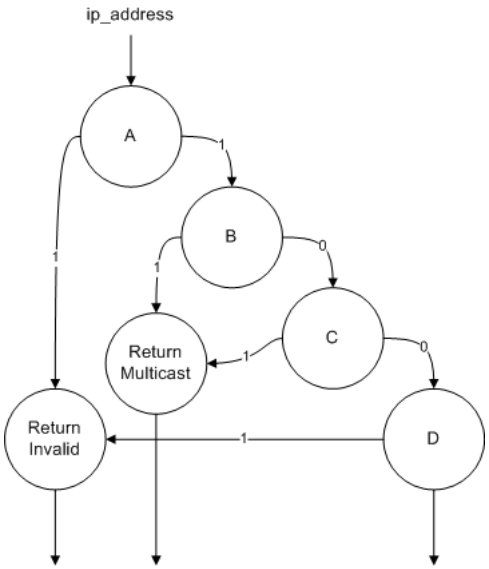


Figure 5.1: Execution Path For Application *n*

the memory space local to the PE. Architecturally, the memory regions reflect the fact that it is common to store packets in one region and control information such as routing tables in another region, while important data is cached close to the PE.

The t_{mem} is therefore given by Equation 5.4.

$$t_{mem} = (N_{pkt} * t_{pkt}) + (N_{ctrl} * t_{ctrl}) + (N_{local} * t_{local}) \tag{5.4}$$

where N_{pkt}, N_{ctrl} and N_{local} are the number of memory operations falling into each region. With the majority of modern RISC architectures following a stack-pointer based architecture, with extensive register movements, it is clear that the number of local memory accesses will be significantly higher than the number of accesses made to either packet or control memory. Since any local memory is assumed to be nested close to the PE, it does not involve any contention or queueing. On the other hand, the control and packet memories are assumed to be shared between all PEs and therefore involve a contention or queue delay along with longer access latency. Wolf and Franklin demonstrated that an M/D/1 queue system provides a good approximation of a shared memory system, particularly at high device loads [135], but does not differentiate the PE, system bus and memory delays. Intuitively, it is possible to deduce the number of memory accesses for certain memory regions under defined conditions. For example, a typical IP forwarding appli-

cation might follow an outline such as; packets are buffered in external DRAM (t_{pkt}), the forwarding table is maintained in shared SRAM (t_{ctrl}) while the packet processing variables are maintained in local per-PE SRAM. On requesting a packet from the queue controller, the PE is allocated the next available packet from a certain queue. Instead of moving packets around memory, the packet request operation results in a pointer to the packet being returned to the PE. Since IP packet forwarding does not require the packet payload to be modified, the PE issues memory read operations to packet memory for the first 20-Bytes of the packet (IP Header) along with any link layer data (interface, time stamp, etc.). Since the updated Time-To-Live and packet checksum must be written back to the header, a reasonable approximation of the number of packet memory operations (per-packet) on a 32-bit architecture is 5 memory reads (1 Word Link Layer Header plus 4 Word IP Header) as well as a single memory write operation. The number of packet memory transactions is therefore $N_{pkt} = 6$. Although the number of forwarding table accesses per packet is unknown, the number will typically be bounded by the underlying algorithm, e.g. a 32-bit multi-bit trie with 4-bit stride will require at most 8 memory operations. In Section 5.3.3 an empirical analysis of memory distribution for NP applications is presented.

5.1.1.3 Instruction Budgets

Recalling the maximum amount of packet processing time available outlined previously in this chapter, there is another method by which NP applications can be examined. Recalling Equation 5.1 it can be seen that, assuming $T_{IR} = T_{process}$, it is possible to calculate the number of instructions which can be executed on each packet in order to maintain line rate. For any microprocessor, the time taken to execute program *n* on packet *p*, the processing time ,$T_n(p)$, is given by:

$$T_n(p) = \frac{N_{ins} * CPI}{f_{clk}} \tag{5.5}$$

where N_{ins} is the number of instructions executed for program *n*, f_{clk} is the processor clock frequency and CPI is the average number of clock ticks required to execute a sin-

gle instruction. While an ideal architecture requires a single clock cycle to execute one instruction, some of the difficulties in obtaining an optimal CPI (1 instruction completing every cycle) were outlined in Chapter 2. Commercial architectures commonly obtain a value in the range of 1.5 to 2 clocks per instruction. Substituting T_{IR} for $T_n(p)$ it is possible to deduce the number of instructions which can be supported during the processing of two minimum sized packets.

$$I_{budget} = \frac{f_{clk} * P_{size} * 8}{LR * CPI} \tag{5.6}$$

For an *n* PE system the instruction budget is scaled by a factor of *n*:

$$I_{budget}(Parallel) = \frac{n * f_{clk} * P_{size} * 8}{LR * CPI} \tag{5.7}$$

In Figure 5.2 the relationship between the instruction budget and the microprocessor *CPI* is plotted for various static parameters. (*LR=622.08Mbps*, *n=8* and f_{clk}*=1GHz*). It can be seen that small increases in the average *CPI* can significantly reduce the instruction budget available for processing. An increase in the average CPI from 1.2 to 1.4 would reduce the instruction budget by ∼800 instructions on a POS link with 46-Byte minimum sized packets [1].

5.2 Workload Analysis

While the analytical framework outlined previously highlights the sensitivity of PE performance to microprocessor *CPI*, the parameters which directly affect *CPI* must be derived from empirical analysis using a workload analysis. The next section presents a brief overview of the applications chosen for this analysis. To ensure a broad analysis, where possible, the algorithms and applications chosen reflect the differing methods by which a function can be implemented. For example, none of the benchmarks proposed for NP architectures include a standard 5-tuple packet classification algorithm. In Chapter 2 the

[1] A 46-Byte POS link assumes 40-Byte minimum sized TCP/IP packets along with a 6-byte SONET control header

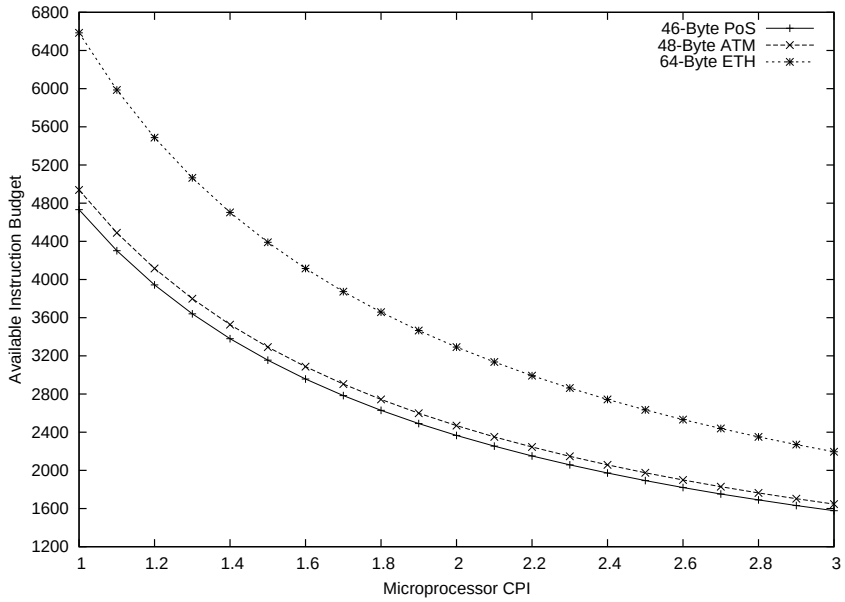


Figure 5.2: Instruction Budget Vs. Microprocessor CPI

importance of classification in a number of network applications was highlighted, but of the various algorithms proposed for 5-tuple packet classification, only HyperCuts and HiCuts share an algorithm similarity. When evaluating NP performance of packet classification it is clear that selecting only a single algorithm would limit any general analysis from being extracted.

With the exception of the structure generation of the Recursive Flow Classification (RFC) [2] algorithm, the remaining code was implemented by the author in C code. Applications obtained from previous benchmarks or public sources have been acknowledged within the research. Applications follow the traditional definition of being either header or payload based functions. From this initial separation, applications are divided into their respective functionality such as IP security, packet classification, packet forwarding, etc. In general, it is unlikely a router would implement only one function, so these applications would typically form building blocks from which the services required by the router would be obtained. Where possible, the implementations match those specifications defined in the relevant Request For Comments. For example, the IP forwarding applications include code to verify the source and destination address, packet header checksum and

[2]For clarity the acronym RFC refers to the Recursive Flow Classification algorithm in this thesis and not the common Internet-based Request for Comments.

the packet Time-To-Live. For the remainder of the thesis, the capitalised abbreviation is used to identify each network application.

5.2.1 Network Algorithms and Applications

Network applications follow the common division of those applications which utilise the packet header only (Header Processing Applications), and those applications which utilise both the packet header and payload during processing (Payload Processing Applications).

5.2.1.1 Header Processing Applications

Packet Forwarding When deploying route lookup algorithms, a network designer is typically concerned with three factors; the lookup complexity associated with performing a single route lookup, the memory requirements needed to store the underlying table structure and the ability to incrementally update the routing table. A large number of algorithms have been proposed which attempt to trade one of these factors off against another. Current solutions can be divided into three categories; hash-based linear searches, trie type structures and hardware accelerated IP forwarding. Each method has a number of advantages when compared. For example, a hash structure typically requires large amounts of memory but guarantees the number of memory operations required to access the routing table.

In addition to route lookup, performance and stability requirements necessitate a number of functions which must be implemented alongside next hop address generation [166]. These packet operations include verifying that the source and destination addresses are valid, checking the header integrity through a checksum operation before finally checking and decrementing the Time-To-Live value. The analysis presented in this work chooses two software algorithms; the Level-Compressed Trie **TRIE**, which implements both level and path compression [167], along with a linear hash based forwarding application in which n routing table entries are mapped to 2^n hash entries **HASH**.

Packet Classification Similar to packet forwarding, a number of algorithms have been proposed for packet classification. Heuristic, Decomposition and trie based algorithms all map the classification ruleset into differing search structures. This work utilises two classification mechanisms, the heuristic **RFC** algorithm [58] and the Hypercuts algorithm **HYPER** [59] for analysis. Both algorithms are optimised for differing performance constraints. While the *RFC* algorithm attempts to minimise the number of memory accesses required to classify each packet, it does require large amounts of memory when compared to the multi-dimensional Hypercuts algorithm.

Traffic Shaping & Queueing With network traffic demonstrating bursty characteristics, some mechanism of shaping traffic patterns to create a more manageable network is required. Metering algorithms allow packets to be marked and processed differently based on the network load at that point in time. Queueing algorithms such as Round Robin (RR), Weighted Round Robin (WRR) and Deficit Round Robin (DRR) allow incoming packets to be fairly distributed across the NP, while algorithms such as Random Early Detection (RED) can be implemented to randomly drop packets if it is clear that minimum service cannot be maintained. Three algorithms in this category are used for analysis in this work. The Token Bucket Metering **TBM** [168] algorithm releases packets at a rate determined by the amount of tokens stored in a bucked at any one point in time. The Two-Rate Three Color Marking **TCM** [169] implements a similar function but can be used to mark incoming packets either Yellow, Green or Red based on the inter-arrival time, packet length and bucket states. Finally, the **DRR** algorithm allows a random data load to be evenly balanced across a number of queues. The queueing mechanism implemented expands a single data source to m unbalanced queues before further expanding the structure to n balanced queues.

Miscellaneous Applications Other functions which do not fall into the categories outlined above include tasks such as statistics gathering or Network Address Translation (NAT). Statistical analysis such as network load, flow-based information or a more finely grained customer-based usage allow network providers to implement functions such as

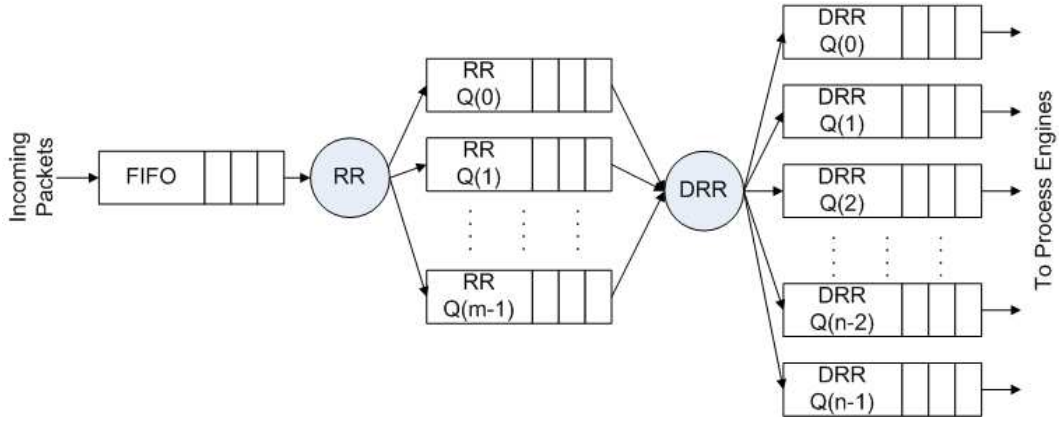


Figure 5.3: Deficit Round Robin Load Balancing

user-based metering and identifying potential bottlenecks. NAT services allow networks to present the same outgoing IP address for all users within the network. At a flow level it can be seen that translation and statistical analysis are similar functions. Both applications use either 2 or 5 tuple classification to identify independent flows within the traffic. The application implemented in this work, **STAT**, implements a 5-tuple flow based statistical analysis. At any point in time the number of active flows is available as well as global statistics such as protocol and port distribution, network throughput, etc.

5.2.1.2 Payload Applications

Packet Encryption Packet encryption is required for functions such as VPN or SSL. Encryption in IP networks is defined by the IPsec cryptographic security suite. To provide secure communication across an open network, IPsec utilises symmetric cryptographic algorithms as a means of encrypting packet data. Algorithm agnostic, IPsec defines the protocol layout without standardising the specific encryption algorithm. At a network level it is possible to trade off security and speed. While block algorithms require extensive processing, a stream cipher allows higher performance to be extracted at the cost of lower security. This work examines three encryption algorithms; the **AES** algorithm [67], the less secure but faster **CAST** algorithm [170], and the stream-based lightweight (but cryptographically weaker) **RC4** cipher [171].

Packet Authentication In addition to packet encryption, the Authentication Header (AH) protocol provides a mechanism for securing and verifying packets sent between networks. For each outgoing packet, a digital signature is created for the packet. This message digest is added to the packet before the modified datagram is encrypted and transmitted. A number of hash algorithms are currently available and can be used within an IPsec implementation, with some algorithms utilising a modified encryption block, while others are built around transforms specifically chosen to be difficult to implement in both forward and reverse directions. The 160-bit **SHA1** [68] and 128-bit **MD5** [172] algorithms are examined in this work.

Error Correction and Detection Network algorithms such as IP or TCP provide a means of verifying that data arriving across a network has not been damaged during transmission, but the checksum employed within TCP/IP headers provides little protection to the header in the case of multiple errors and provides no means of recovering packets if re-sending is not possible or guaranteed. A more comprehensive method of verifying data is to employ a software based error detection algorithm which allows the entire packet to be checked for multiple errors. At transmission, the checksum code is generated and transmitted with the packet. Once the packet is received, the checksum generation is repeated to verify both values. Should packet correction be required, a more computationally intensive algorithm can be utilised. For a large proportion of IP based traffic it can be argued that complex error correction codes are not required since any corrupted packet can always be resent at little cost. But for certain networked cases it may not be possible to resend data with error correction codes common in satellite, wireless and broadband network technologies. For technologies highly sensitive to latency, such as video streaming using the H264 Scalable Video Codec [173], routing-based forward error correction becomes ever more important, with Cisco's ASR9000 Routers providing optional error correction [174] at the edge-level of the network. This workload analysis uses two algorithms, the simple 32-bit error detection **CRC** algorithm and the Reed-Solomon [175] error correction algorithm **RS**.

Packet Manipulation In addition to packet security and verification, another payload function commonly implemented at a router level is packet manipulation. An example of these manipulation routines can be seen in the fragmentation algorithm. Since not all networks provide the same performance or even follow the same level 2 specification, the maximum packet size allowed on a network may vary from one to another. To deal with this situation, routers typically fragment large packets into smaller blocks at the entry point of the network, while the routers employed on the larger Maximum Transmission Unit (MTU) network might implement a re-assembly algorithm which attempts to rebuild fragmented packets arriving to this network. This workload analysis uses this fragmentation algorithm **FRAG**.[3]

5.2.2 Simulation Parameters

Table 5.2: Summary of Applications Analysed

Function	Key	Function
Packet Forwarding	TRIE	LC-Trie forwarding
	HASH	Linear Search Forwarding
Packet Classification	HYPER	Trie-based 5-Tuple Classification
	RFC	Heuristic 5-Tuple Classification
Packet Metering	TBM	Token Bucket Metering
	TCM	Two-Rate Three Color Marker & Metering
Queuing	DRR	Deficit Round Robin
Miscellaneous	STAT	Statistics & Flow Maintenance
IPsec Encryption	AES	128-bit AES-CBC
	CAST	128-bit CAST-CBC
	RC4	Stream Cipher
IPsec Authentication	SHA1	160-bit Message Digest
	MD5	128-bit Message Digest
Error	CRC32	32-bit Error Detection
	RS	Reed-Solomon Error Correction
Manipulation	FRAG	Packet Fragmentation

[3]The fragmentation algorithm is expected to be phased out at router level as IPv6 gradually replaces IPv4.

Utilising the SimNP simulator described in Chapter 4, the simulation parameters used for the analysis presented in this chapter are first outlined. A summary of the algorithms examined in this work is given in Table 5.2. For the forwarding applications, routing tables are derived from common backbone network connections, with the TRIE algorithm using the 117,000 entry AT&T forwarding table and the HASH algorithm using a smaller 27,000 entry MAE-WEST routing table. For the classification algorithms, rulesets derived from Classbench [176] are used. Both classification applications use a 1000 entry ruleset, with the simulation traces modified to match the semi-synthetic rulesets. In addition to the OC-12 (AMP) and OC-48 (PSC) traces utilised in Chapter 4, a slower OC-3 TXS trace is also used during simulation. As with the AMP and PSC traces, the TXS trace is modified since NLANR trace files do not include valid IP addresses. The trace files are summarised in Table 5.3. For analysis which is PE specific (e.g. instruction distribution), SimNP is configured using a single PE with a relative latency between the PE and memory of 5 cycles (both packet and control memory). When examining the degree of parallelism the PE to memory latency is fixed at 10 cycles as the number of PEs is increased. Each PE is configured with a coupled zero-latency 128-KByte local SRAM. For branch behaviour analysis the underlying system parameters are not important since branch penalties or even predictor performance (examined in Chapter 6) is independent of parameters such as memory latency or bus utilisation.

Table 5.3: Summary of Trace

Trace	**POS Connection**	**N$_{pkt}$**	**L$_{avg}$**	**TCP %**	**UDP %**	**OTH %**
TXS	OC-3	17,000	90	80.41	8.1	11.49
AMP	OC-12	250,000	875	96.87	2.28	0.84
PSC	OC-48	1,000,000	704	86.46	10.75	2.78

5.3 Simulation Results

The simulation results are divided into a number of sections. Firstly, the instruction distribution and instruction budget decide whether an application could be supported under realistic conditions. Secondly, those aspects which reduce PE utilisation are examined.

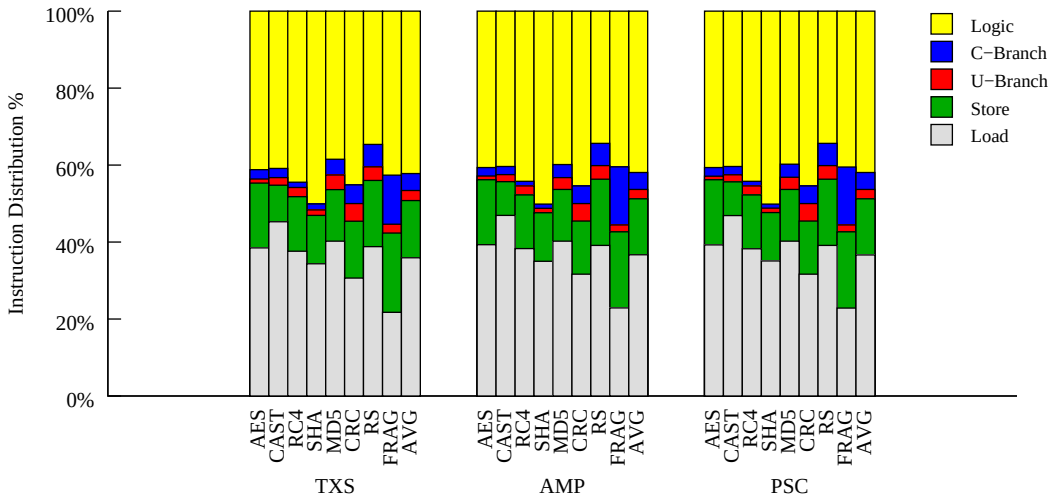


Figure 5.4: Instruction Distribution for NP Payload Applications

The parameters include the memory footprint of NP applications, the degree of PE parallelism employed in a shared bus system and finally the performance penalty associated with conditional branch instructions within NP applications.

5.3.1 Instruction Distribution

The first aspect of any workload analysis is to classify the types of instructions commonly used in network applications.

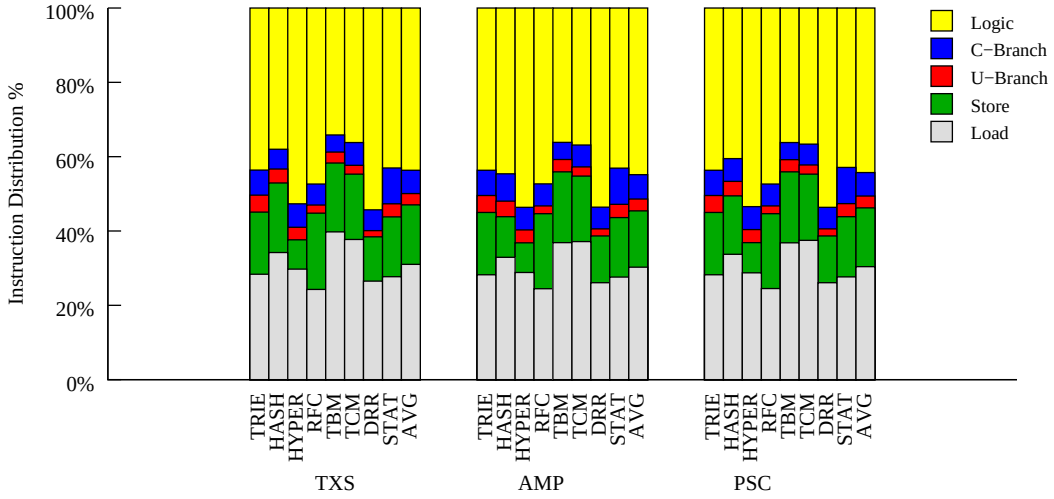


Figure 5.5: Instruction Distribution for NP Header Applications

The high memory usage of NP applications is demonstrated in Figure 5.4 and Figure 5.5 which summarise the instruction distribution across three separate traces. Averaged

across all three traces, memory load and store operations comprise 46.2% of header applications and 51% of payload tasks and highlight that memory latency hiding techniques such as multi-threading will remain important methods of ensuring high PE utilisation, especially with memory access speed failing to keep pace with CPU performance. It can also be seen that the number of branch operations is higher for header applications than payload functions, an average 9.5% of all instructions for the 8 header applications and 7% of the payload applications. Both of these topics are covered in more detail later in this chapter.

5.3.2 Instruction Budget

In Section 5.1.1.3 the instruction budget metric was defined in terms of available processing time per packet. For a given PE CPI, it is possible to estimate how many instructions can be executed while maintaining the required wire speed. Previous workload analyses have attempted to link this instruction budget to the average instruction count per packet. While the average instruction count provides a good snapshot of the workload complexity of an application, it is not possible to determine if a given application can be supported via the average instruction count. As an example, consider an IP forwarding application. In this case, the instruction budget available to one packet must be less than the maximum instruction count of the application. For a payload application, the instruction budget is defined by the number of instructions required to process two minimum sized packets arriving at the line card back-to-back.

5.3.2.1 Header Application Instruction Budget

While a relationship between the instruction count per packet and the payload length can be deduced for payload based NP applications, it is not possible to extract a similar model for header based applications. In general, the processing time will be determined by the control variables used during execution. For example, the number of read operations required to traverse a trie structure depends on the number of memory reads per node as well as the initial key being searched for within the trie structure. Examining the per

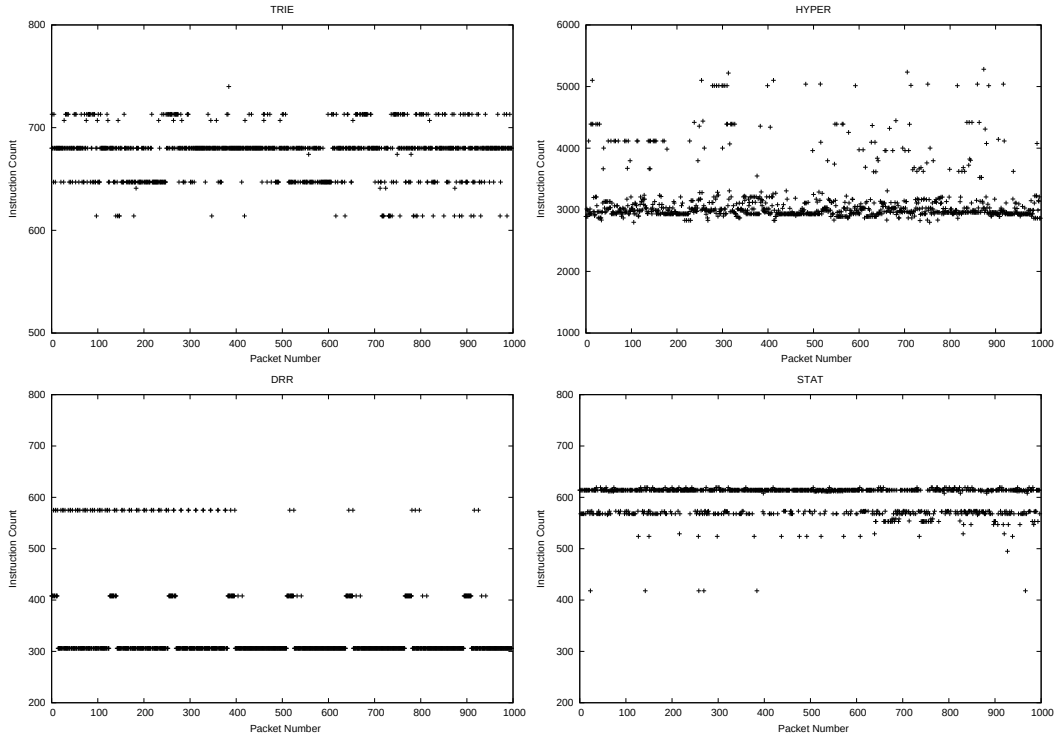


Figure 5.6: Header Application Instruction Count Per Packet Distribution

packet instruction traces for four NP header applications in Figure 5.6 (TRIE, HYPER, DRR & STAT) it can be seen that, with the exception of the HYPER algorithm, there are only a small number of possible execution paths. In the case of TRIE based IP forwarding there are six possible execution paths ranging between 614 and 713 instructions, with an examination of 50000 packets finding that three execution paths (647, 680, 713) cover 93.76% of all processed packets. The STAT and DRR metering algorithms exhibit similar structures, with only 3 processing paths for DRR and the three most common paths for the STAT algorithm covering 77.34% of all packets. In Table 5.5 and Table 5.6 the average, minimum and maximum instruction counts for the TXS and PSC traces are given.

It was seen in Equation 5.7 that to support a given application the number of cycles used to process a packet must be less than the packet inter-arrival time at that line rate. For header applications the difference between maximum processing time and the average processing cost varies from almost zero for HASH-based IP forwarding to over 63% for the Hypercuts classification algorithm. Across all header applications the results highlight the importance of determining the maximal processing cost during analysis. In Chapter 2 it was seen that, with the exception of the Intel IXP, it was common for NP architectures

to include some form of packet metering hardware. As can be seen in table 5.5, the TCM, TBM and DRR algorithms all require at most 575 instruction cycles to deploy and are small enough to be implemented on a 1GHz PE, processing packets at up to OC-12 POS line rates.

To evaluate which applications could be supported and at what degree of parallelism an instruction budget model is constructed. The PEs utilise a 1 GHz 5-stage ARM9 type processor, with a *CPI* of approximately 1.5 [177]. A standard TCP/IP network is used with a minimum packet size of 40-bytes. The instruction budgets for OC-3, OC-12 and OC-48 optical links are are shown in Table 5.4.

Table 5.4: ARM9-Type Instruction Budget

Interface	$\mathbf{T}_{IR}$	$\mathbf{I}_{budget}$
OC-3	2000	1371
OC-12	514	342
OC-48	128	85

Table 5.5: Instruction Complexity (Header Applications)

Application	$\mathbf{I}_{pp}$**(Avg)**		$\mathbf{I}_{pp}$**(Min)**		$\mathbf{I}_{pp}$**(Max)**		δ **%**	
	TXS	PSC	TXS	PSC	TXS	PSC	TXS	PSC
TRIE	684	675	614	395	713	746	4	9.51
HASH	4613	4613	4613	4614	4624	4624	0.2	0.2
HYPER	3575	3125	2828	2763	5254	5121	31.9	63.8
RFC	492	524	286	296	531	531	7.3	1.3
DRR	365	504	306	306	575	575	36.5	12.34
TCM	259	215	206	206	261	253	0.7	15
TBM	170	152	152	152	178	174	4.5	12.6
STAT	531	563	392	347	619	619	14.2	9.0

In Table 5.5 the actual instruction complexity for each header application is obtained from simulation. In addition to the average instruction count per packet (I_{pp}(Avg)), the more useful maximum (I_{pp}(Max)) and minimum (I_{pp}(Min)) instruction counts are also extracted. The final column in Table 5.5 highlights the importance of extracting maximal processing costs from any workload analysis, with the average processing cost for the HYPER algorithm understating the processing cost by nearly 64% across the exact

same packet. It can be seen that a single PE system provides enough computing resources to maintain an OC-3 connection across all applications except the HASH and HYPER based algorithms. While for an OC-12 based router the number of PEs must be increased to three in order to support applications such as TRIE-based forwarding, RFC classification, DRR-based queueing or a flow based statistical analysis. Increasing the line rate to 2.4 Gbps (OC-48) reduces the instruction budget to only 85 instructions, highlighting the need to minimise the average CPI in order to meet future demands. With a budget of 85 instructions per packet, IP forwarding would need to be divided over approximately 9 PEs for peak line rate to be maintained. It should be noted that the HASH based implementation is not optimised and it may be possible to significantly improve performance by implementing either a more refined hash algorithm or by providing each PE with a small hash generating engine. While the Hypercuts algorithm is more efficient in terms of memory usage, its computational complexity is on average 7 times greater than the RFC algorithm. Furthermore, only $\sim$ 180 instructions are actually related to the classification algorithm, while the remaining instructions are required to fetch and extract the packet tuples.

5.3.2.2 Payload Application Instruction Budget

As expected, for payload applications, the number of instructions executed on an *n*-byte packet will increase linearly with changes in the packet length (Figure 5.7). Utilising the base α plus length component β estimation outlined in [178], the processing time for payload applications can be assumed to be:

$$T_n(p) = \alpha + \beta * P_{size} \tag{5.8}$$

Analysis of simulation data allows the values for α and β to be calculated (Table 5.6). With the exception of the hashing algorithms, the initial processing cost is quite small when compared to the per byte instruction count. For example, IPsec *AES* encryption of a zero byte packet requires 326 instructions (α) to be executed while encryption of a 4 byte string requires 676 instructions to be executed.

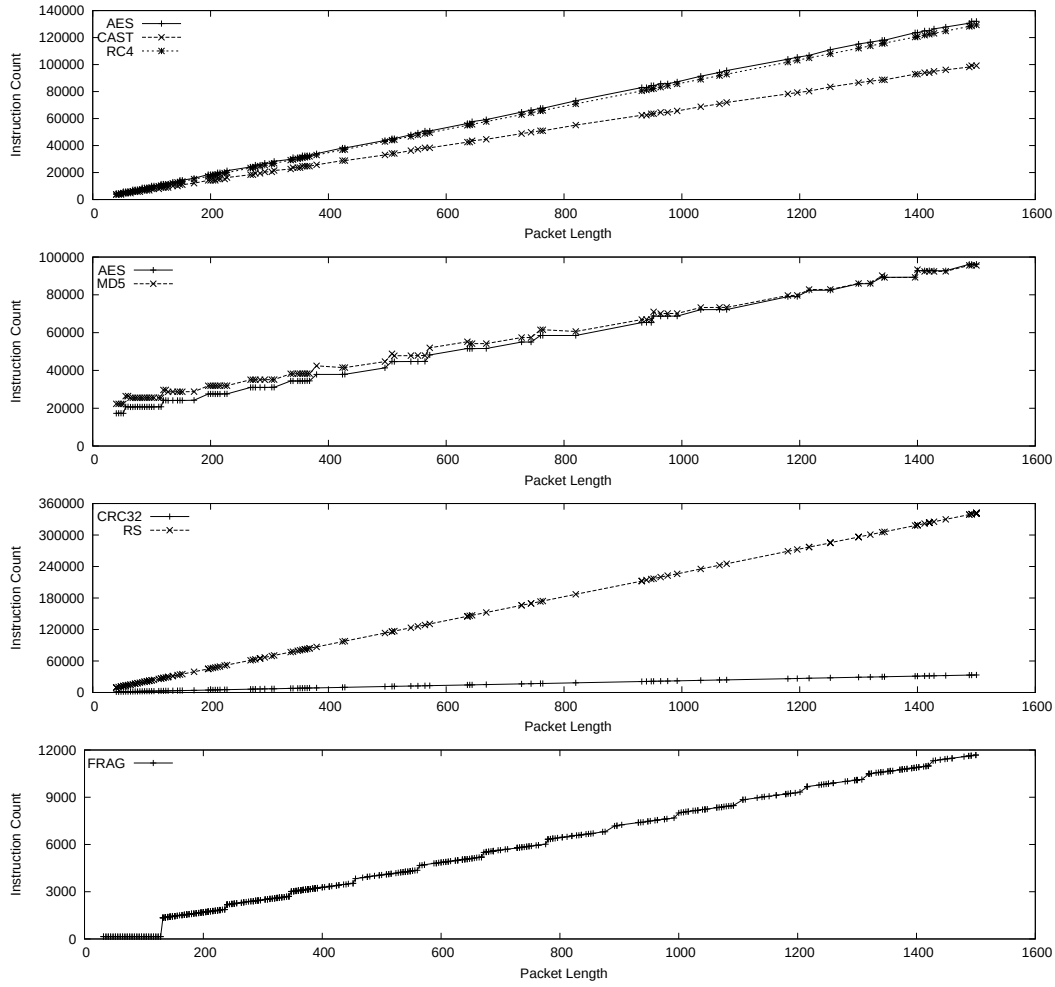


Figure 5.7: Instruction Count Vs. Payload Length

For payload applications the worst case scenario is when two minimum sized packets arrive back-to-back at the network interface. Across all security algorithms it is clear that a high degree of parallelism would be required to support each application in a fully software based implementation. Again, assuming a similar PE configuration to that used in the previous section ($f_{clk} = 1GHz, CPI = 1.5$), the AES, CAST and RC4 would all require between 7 and 10 PEs running in parallel. The high initial time required to configure a software based hashing algorithm would seem to make such implementations too expensive to deploy. Discounting the per-packet base instruction cost (α), the per-byte instruction costs of 244 and 231 (Table 5.6) would require 9760 and 9240 instructions for the SHA and MD5 algorithms respectively. Without massive increases in parallelism it is difficult to see error correcting codes such as the Reed-Solomon encoder being deployed on software-based NP systems.

Table 5.6: Instruction Complexity (Payload Applications)

Application	α	β	I_{pp}(Avg)		I_{pp}(Min)		I_{pp}(Max)	
			TXS	PSC	TXS	PSC	TXS	PSC
AES	326	87.5	9173	62455	3128	3128	132020	137624
CAST	525	130	7156	47787	2625	2626	99225	99225
RC4	324	86	8439	61707	2732	3076	129324	129324
SHA	7591	244	20666	53899	17281	17281	96201	96201
MD5	13079	231	25572	56363	22309	22309	95603	95603
FRAG	146	5.5	579	5440	146	146	11681	11681
CRC	395	22	2474	15861	1011	1011	39896	39896
RS	112	232	21928	160369	6794	6794	509165	512764

5.3.3 Memory Distribution

Examining published work, it is clear that memory latency remains a substantial issue regarding PE performance. The high proportion of memory operations places large demands on the bus and memory hierarchies which connect the PE array to external memory. Segmenting memory into the three regions (control, packet and local), it is possible to determine how each memory region affects PE (or NP) performance. As an example, consider a PE executing IP packet forwarding. Once allocated a packet the PE must read 5 data words from packet memory (4-Byte control structure + 20-Byte IPv4 header). From the previous workload sections it is possible to see that on average the PE will execute 684 (Table 5.5) instructions during processing (TXS-Trace), of which 45.08% are load and store instructions ($\sim$ 307 instructions). For a 32-bit architecture the packet memory functions would require 5 data loads and a single data write. With the TRIE algorithm configured so that the maximum number of trie lookups is eight, with 2 additional memory reads required to fetch the node structure, the total number of N_{pkt} and N_{ctrl} operations is 6 and 10 respectively. The remaining 292 memory instructions are therefore accesses to the local memory and are primarily used for stack operations (PUSH/POP) and data movements between the register bank and local memory.

Across all NP applications examined in the work it is possible to deduce a similar result with the exception of the fragment function. In Figure 5.8 and Figure 5.9 the memory distribution for each application is shown. The PE was configured with a small amount of

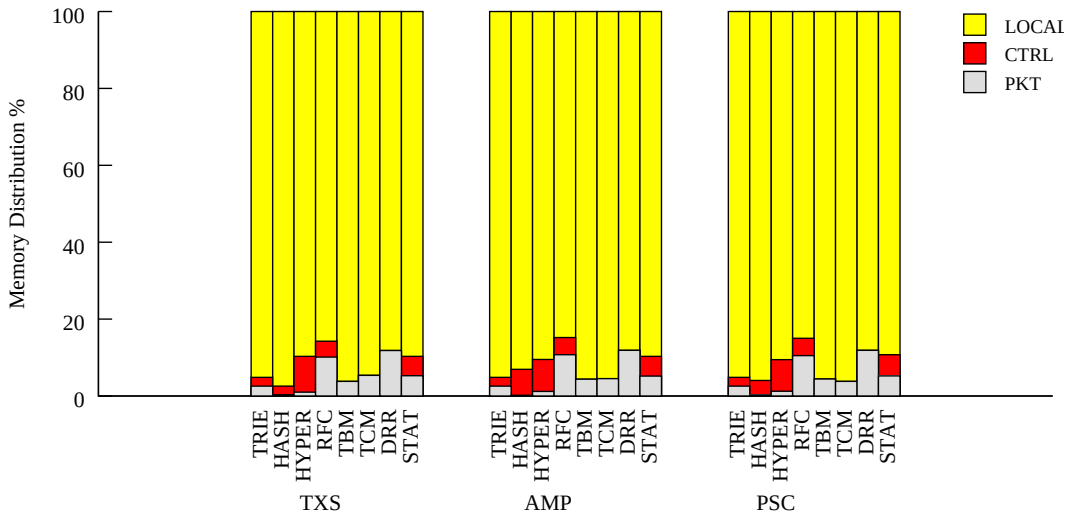


Figure 5.8: Memory Region Distribution (Header Applications)

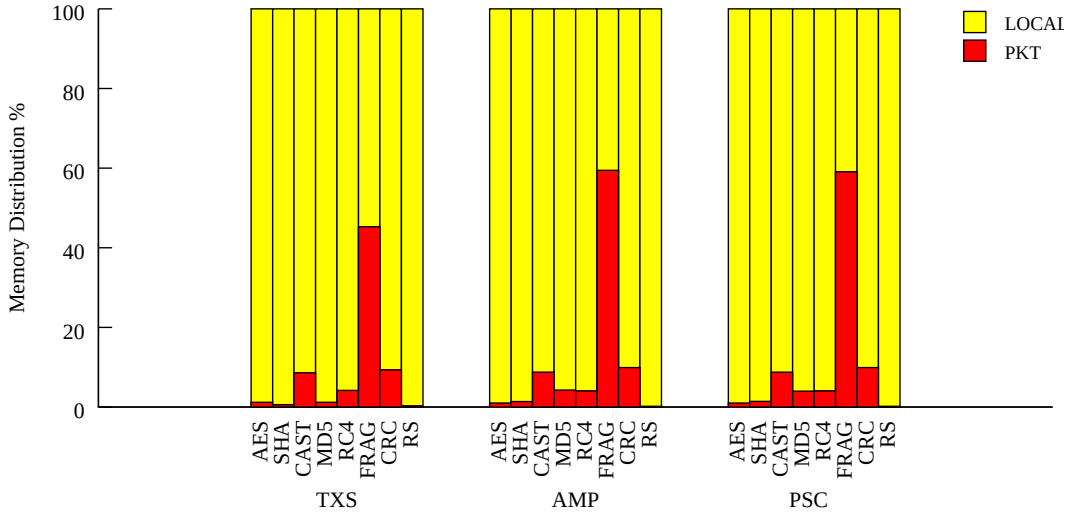


Figure 5.9: Memory Region Distribution (Payload Applications)

on-chip per-PE local SRAM (65 Kbytes). This memory maintains program control structures (stack, heap and initialised data) and variables swapped out of the register bank. For header applications the number of memory accesses to local memory is almost 92% across all three traces, with only DRR and RFC generating packet memory requests in excess of 10% (11% and 10%). For the TRIE algorithm, the figures are 2.5% (8 operations) to packet memory, 2.3% (7 operations) to control memory and 95.1% to local SRAM.

Since none of the payload applications require access to shared control memory, the memory requests are divided between packet memory and local per-PE memory. Despite each application having to transfer the entire payload back and forth from packet memory, the proportion of packet memory requests remains significantly smaller than local mem-

ory accesses. Part of this is due to the fact that security algorithms such as AES or SHA are factored to operate on a block size buffer, requiring stack operations (Load/Store on an ARM architecture) to be called during each sub-routine jump and return. The majority of local memory accesses are because operations involving large blocks of data in parallel will tend to saturate the PE register base, requiring state variables to be constantly swapped between the PE register base and local SRAM. Across all 8 applications, only the FRAG application requires a high proportion of packet memory operations (either read or write). This is an expected result since it does not actually make significant changes to the packet, simply moving the data from one buffer to another. The results highlight that one method of improving PE utilisation is to ensure that only a small latency between the PE and local memory is present. Additional register space will also help to reduce the number of swap instructions required during processing.

5.3.4 Parallelisation

The limitation on PE performance due to parallelism is examined in this section. The negative effects of parallelism can be examined by determining either the NP stall rate or the per-packet processing rate for each PE. In the first case, the stall rate can be defined as the number cycles when all PEs are stalled while waiting for external operations (either memory or interface) to finish. The per-packet processing rate is the average number of cycles (c) required for a single PE to process a single packet. An n PE system would ideally require c/n cycles to process the same packet. The deviation away from the ideal provides a mechanism for determining the degree of parallelism which can be supported. Commonly referenced as Amdahl's law, it provides an empirical method of determining the expected speed-up which can be achieved via increased parallelism.

5.3.4.1 Stall Rate

The utilisation rate for each PE can also be calculated as the ratio of active cycles to total cycles. In Figure 5.10 the stall rate for each PE, as well as the stall rate for the entire processing array, is given for four applications. The stall rate for the processing array

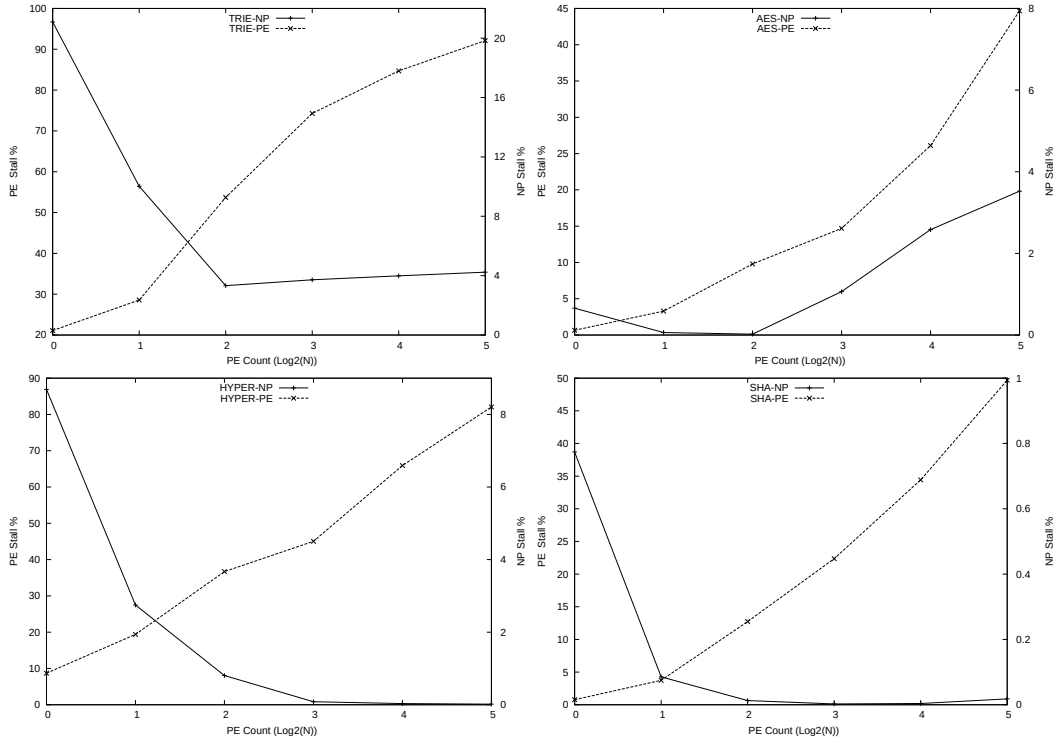


Figure 5.10: NP System Stall Rate(Modified Round Robin Bus)

(NP Stall Rate) is the proportion of the device cycles when all PEs are inactive while waiting for external data. Above eight PEs the average utilisation rate for each PE is below 50% for both header based applications. In the case of IP forwarding, each PE is stalled on average 92% of the time when 32 PEs are used in parallel, highlighting that increased parallelisation may not be possible without significant increases in memory and bus performance. It should be noted that the above simulations actually underestimate the degree of parallelism available since it was assumed that external memory operated at the same frequency as the PE array. The utilisation rate for the HYPER algorithm increases linearly from only 10% for a single PE system to $\sim$82% for a 32-PE system. With payload applications employing a large number of ALU instructions during packet processing the utilisation rate is higher for both single and multiple PE systems. In fact, even scaling the number of PEs running either AES or SHA would continue to yield positive PE utilisation rates ($\geq 50\%$). The NP stall rate increases from 1.06% to 3.5% as the number of PEs is increased from 8 to 32, indicating that device contention (bus and external memories) becomes a performance bottleneck at higher levels of parallelism, despite the long processing times between packet transfers to packet memory.

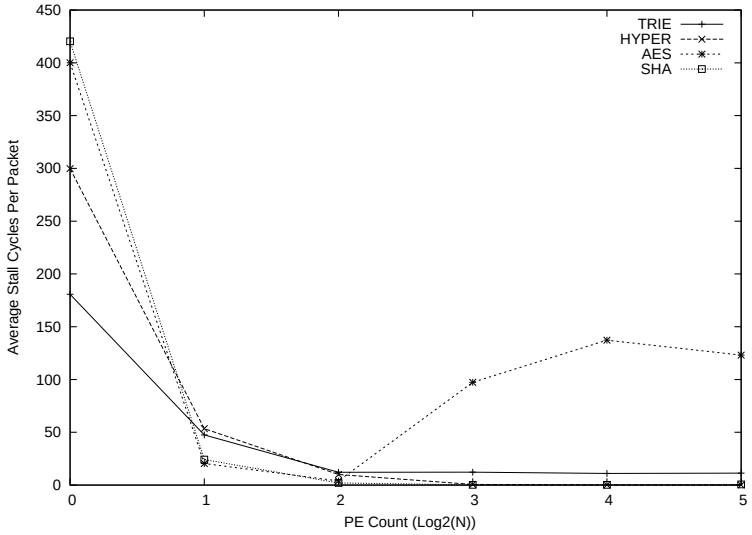


Figure 5.11: NP Stall Cycles Per Packet

The NP stall rate can also be quantified as a per-packet penalty applied to each packet traversing the router. For a single PE system, $\rho_{pe} = \rho_{np}$, the fact that commands are not interleaved between parallel PEs results in a high per packet penalty. As the number of PEs is increased the per-packet NP stall rate decreased before increasing once contention becomes an issue. In Figure 5.11 it can be seen that the stall penalty per packet falls to 11 cycles for TRIE, 1 cycle for HYPER, 1 cycle for SHA and 4 cycles for AES but increases rapidly for AES when more than 4 PEs are employed.

5.3.4.2 Per-Packet Processing Rate

Another method of analysing parallelism involves quantifying the average number of cycles per packet. When using Amdahl's law to estimate the speed-up due to parallelisation, two factors within the program must be determined. Firstly, *P* is defined as the proportion of the program which can benefit from parallelisation by degree *N*, while *1-P* is the proportion of the program which cannot be parallelised. The speedup is therefore:

$$S = \frac{1}{1 - P + \frac{P}{N}} \tag{5.9}$$

For an NP system it was commonly found that packet *p* is independent of other packets within the stream, removing any dependencies between one PE and another. Since a

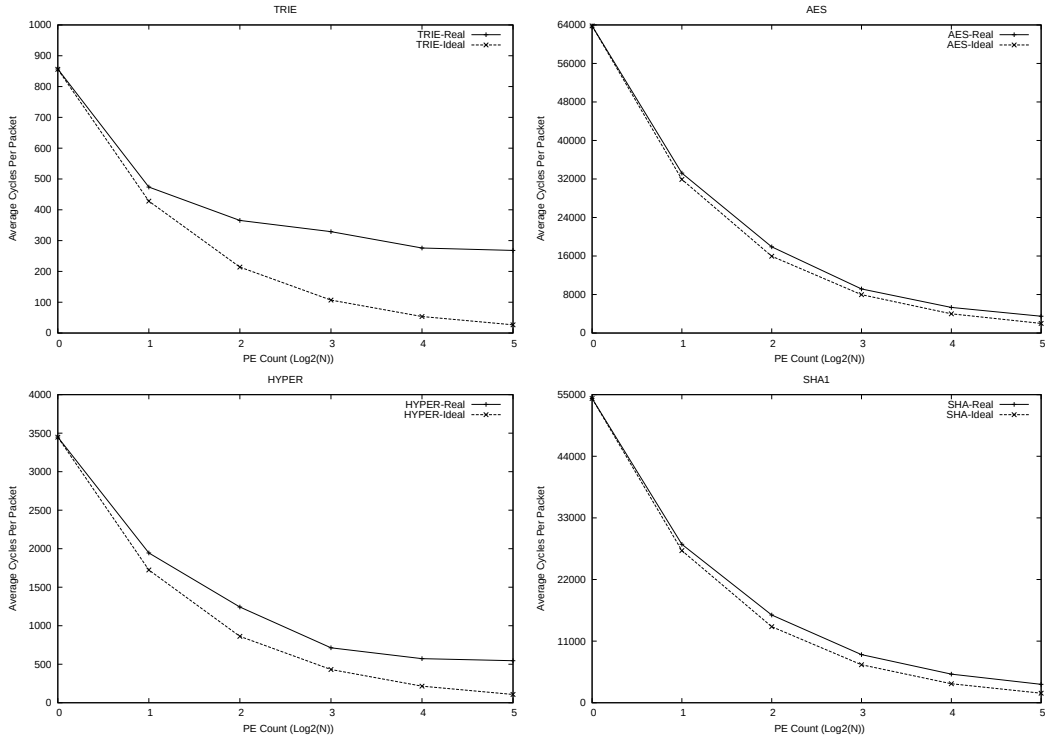


Figure 5.12: Per-Packet Processing Rate

parallel system such as an NP involves some degree of contention the performance increase is scaled by the PE utilisation ρ_i. As the degree of parallelism increases the PE utilisation will fall since the majority of the PE time will be spent waiting for external device access. In Figure 5.12 the per packet rates for the four applications are shown alongside the ideal processing rate ($\rho_i = 1$). As can be seen, the two header applications deviate more significantly from the ideal when compared to the AES and SHA algorithms. For IP forwarding, the per packet processing rate reduces by 44.7% when *N* is increased from 1 to 2 but only 2.5% when *n* is increased from 16 to 32. Similar to previous results, the small amount of ALU instructions relative to the amount of memory operations limits the degree of parallelism which can be employed, with the cycles per-packet processing rate instruction only 13 clock cycles less for a 32 PE system when compared to a 16 PE system. For the HYPER algorithm the processing rate levels off at 8 PEs. The processing times for both payload applications continues to decrease even for a 32 PE system. Despite having a much lower application complexity, both header applications suffer from diminishing returns for high levels of parallelism.

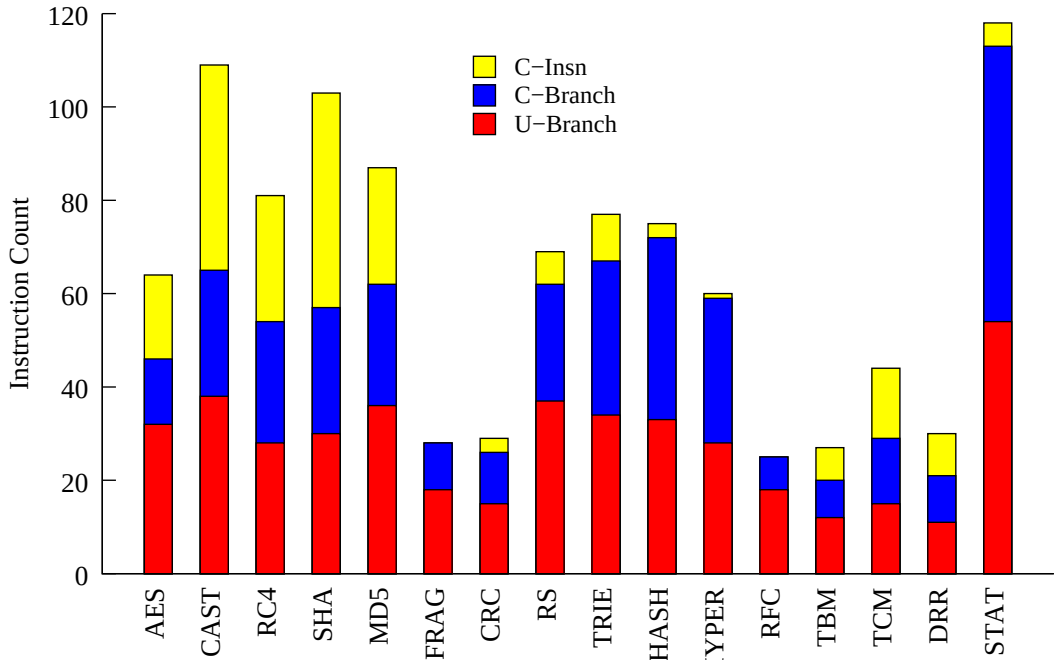


Figure 5.13: Static Branch Analysis of NP Applications

5.4 Conditional Branches within NP applications

In addition to either bus contention or memory bandwidth limitations, another parameter which determines PE utilisation is the branch behaviour of NP applications. As was discussed in Chapter 2, each branch operation requires either the processor pipeline to be stalled while the branch is evaluated or for some means of predicting if a branch will be taken to be implemented. In this section a detailed analysis of branch behaviour in common NP applications is presented. It should be noted that the use of an ARM architecture minimises the total number of conditional branch instructions since all instructions can be made conditional within the ARM ISA, allowing certain small *if-else* statements to be reduced to a single conditional ALU operation. This feature accounts for the some of the difference between the instruction characterisation presented in this work, which found conditional operations account for 8.5% of instructions, while previous work in [100] and [91] found that branch instructions comprise 18% and 16.2% of typical network workloads, on different ISAs.

5.4.1 Static Branch Analysis of NP Applications

In Figure 5.13 a static branch analysis for various NP applications is presented. Each application is compiled as a static binary using an ARM-targeted cross compiler (*gcc-3.4.3*). The *U-Branch*, *C-Branch* and *C-Insn* fields refer to the number of unconditional branches, conditional branches and conditional ALU operations within the object code. While some of the conditional operations are related to how the ARM architecture handles function calls and sub-routine returns during execution, it should be noted that when extending this analysis to a more generic framework some of these conditional instructions would be translated into *traditional* conditional branch segments, i.e. branch operations over small sections of code. As was noted previously, while the applications are designed to be as realistic as possible, some of the processing conditions are simplified for simulation purposes. For example, the IPv4 forwarding algorithm verifies each packet before calculating the next hop address. A more complete PE implementation would include a method for detecting network control packets, such as ICMP packets used for inter-router communication. Within a typical router framework it is unlikely that data-plane PEs would be charged with processing such packets, with the control plane processor providing better mechanisms. As such, other processing paths and subroutines would be required in order to create an implementation-ready application.

5.4.2 Dynamic Branch Analysis of NP Applications

An examination of the simulation traces allows dynamic branch behaviour to be extracted. In general, a dynamic analysis allows branch behaviour to be classified at run time. For example, while the static analysis identified the number of conditional operations and allows the application paths to be formed into a tree-type structure, which can be analysed at compile time, the absolute number of branch instructions is not useful if a single branch loop is called multiple times. For an NP system, the branch metrics are defined in conditional branches per packet. In Table 5.7 the dynamic analysis for the 16 NP applications is shown. The first two columns represent the percentage of the instruction workload which is comprised of unconditional branch instructions (**U-BR**$_d$) and condi-

Table 5.7: Dynamic Branch Analysis of NP Applications

Application	**U-BR**$_d$	**C-BR**$_d$	N_{br}***(min)***	N_{br}***(max)***	$\rho_{\mathbf{tk}}$
AES	1.03	2.45	90	2942	0.65
CAST	1.98	2.57	95	2119	0.12
RC4	2.37	1.37	53	1521	0.03
SHA	1.45	1.59	290	893	0.30
MD5	3.81	4.11	944	3046	0.10
FRAG	2.29	12.63	10	1775	0.27
CRC	4.55	4.89	54	1526	0.12
RS	3.55	5.82	559	19631	0.38
TRIE	4.56	6.77	44	48	0.22
HASH	3.91	6.62	234	240	0.37
HYPER	3.54	6.18	150	353	0.29
RFC	2.15	5.76	11	32	0.6
TBM	3.29	2.93	8	9	0.47
TCM	2.48	6.44	11	17	0.55
DRR	1.9	5.8	14	33	0.73
STAT	3.58	9.74	35	60	0.35

tional branch instructions (**C-BR**$_d$). Examining the instruction flow on a per packet basis it is possible to determine the maximum and minimum number of conditional branches per packet (**N**$_{br}$**(min)**, **N**$_{br}$**(max)**). Finally, it is important in quantifying the performance penalty associated with branch operations to determine the ratio of taken to not taken branches within the application (ρ_{tk}). Typically defined by both the underlying application and compilation algorithm, the ratio of taken branches allows only those branches which incur a pipeline flush (or stall) to be included in the analysis. Similar to the number of instructions per packet, for payload tasks the number of conditional branches is largely determined by the packet length, although certain payload applications such as DPI would not fall into this category.

When considering static branch prediction techniques, the proportion of branches taken highlights the challenge with most static analysis methods. Consider the two IPsec encapsulation applications, AES and CAST. While 65% of branches are taken during AES encryption, only 12% of branches are taken during CAST encryption. Across all

applications the proportion of taken branches ranges from only 3% for the RC4 algorithm to almost 75% of branches for the DRR algorithm. An *at-compilation* heuristic approach would therefore have to apply generic rules to a wide variation of algorithms. On the other hand, for payload applications, any scheme which attempts to maintain a run-time history of branch decisions must take into account the length distribution commonly seen in Internet traffic, with packets less than 100 bytes and greater than 1000 bytes making up the majority of Internet traffic.

5.4.3 Branch Penalty per Packet

Recalling Section 3.3.2.1 the average branch penalty per packet can be calculated using statistics gathered from simulation. With no branch prediction scheme employed, the penalty in cycles lost per packet for each application is shown in Table 5.8. For a minimum sized 40-Byte packet encrypted using the AES algorithm, 293 processor cycles are lost due to taken branches, while for a maximum sized 1500-Byte packet the branch penalty is almost 3000 processing cycles. Recalling the number of instructions per packet outlined in Table 5.6, the number of instructions required to encrypt a minimum sized packet is 3128, similar to the branch penalty incurred for a 1500-Byte packet. With Internet traffic following a complex distribution, in which large packets make up a large proportion of IP traffic and most applications operate in a greedy mode, a penalty of one minimum sized packet for each 1500-Byte packet would be difficult to ignore. For header applications the branch penalty is smaller but remains a significant loss of processing capabilities. For TRIE-based forwarding, the branch penalty is at least 48 cycles per packet or 7% of the instruction count (Table 5.6).

5.5 Conclusions

While previous research has examined NP workloads, with comparisons to general purpose applications, the analysis presented in this chapter attempted to determine and quantify those factors which determine PE utilisation. With maximum performance in a

Table 5.8: Branch Penalty Per Packet ($\rho_{tk} = 5$)

Application	$\rho_{\mathbf{tk}}$	N_{br}***(min)***	N_{br}***(max)***	$\tau_{\mathbf{tk}}(\mathbf{min})$	$\tau_{\mathbf{tk}}(\mathbf{max})$
AES	0.65	90	2942	293	9562
CAST	0.12	95	2119	57	1271
RC4	0.03	53	1521	8	228
SHA	0.30	290	893	435	1340
MD5	0.10	944	3046	472	1523
FRAG	0.27	10	1775	14	2396
CRC	0.12	54	1526	32	916
RS	0.38	559	19631	1062	37299
TRIE	0.22	44	48	48	53
HASH	0.37	234	240	433	444
HYPER	0.29	150	353	218	512
RFC	0.60	11	32	33	96
TBM	0.47	8	9	19	25
TCM	0.55	11	17	30	47
DRR	0.73	14	33	51	120
STAT	0.35	35	60	61	105

pipelined PE achieved when the pipeline remains full, pipeline bubbles or stalls can significantly decrease performance. Using the SimNP simulator outlined in Chapter 4 the analysis in this section determined what applications can realistically be supported on a programmable PE platform for various network line rates. While software based implementations of security algorithms would require massive degrees of parallelism in order to support high bitrates, header applications are typically small enough (in terms of instruction count) to be implemented in software. In general, NP applications require a high proportion of memory instructions (relative to the ALU instructions), highlighting possible limitations in the degree to which parallelism can be used to increase NP performance. Even for a PE design with zero latency local SRAM, the need to access external control and packet memory will significantly affect PE utilisation. It is clear that high levels of parallelism can only ensure future performance gains with corresponding improvements in both bus and memory technologies. Without significant improvements in these components, high degrees of parallelism will quickly result in PE under-utilisation. In the case

of the configurations examined in this chapter, 8-16 PEs is found to be the optimum for header based applications while up to 32 PEs can be configured in parallel for payload based functions.

While memory access latency and bus contention represent two external sources which reduce PE utilisation, the effect of branch instructions within NP applications represent an 'internal' PE performance limitation. Both the analysis in this work and previous workload analyses have highlighted the high percentage of conditional operations within NP applications. For a deeply pipelined PE the effect of these conditional operations is to result in a large amount of wasted cycles after only a short period of time. Unlike general purpose systems which have input sources as varied as network interfaces to keyboards, an NP platform operates on packets only, with the same application remaining in place for long periods of time. With this is mind it should be possible to minimise the amount of processing cycles lost due to branch operations. By taking into account some network traits, it is believed that prediction methods, specific to PEs, should be able to significantly reduce this branch penalty.

CHAPTER 6

Branch Prediction in Process Engines

6.1 Introduction

Following on from the workload and branch analysis presented in Chapter 5, this chapter presents a detailed examination of existing branch prediction schemes when applied to network workloads. In each case, the existing predictor architectures were implemented as simulation models within the SimNP simulator. In Chapter 3 the metrics used to evaluate branch prediction architectures were discussed. In general it is possible to model branch prediction at a relatively high functional level, with branch prediction evaluation and analysis well suited to the SimNP platform. Examining existing prediction schemes it is found that no current method fully exploits the unique nature of NP applications, providing scope for a new NP-specific prediction mechanism. Whereas existing prediction schemes aggregate branch history via a number of saturating counters in order to guide future predictions, the field-based scheme proposed in this chapter attempts to incorporate a number of NP specific traits as a means of improving prediction performance. This new field-based prediction architecture is described before a detailed performance evaluation of the scheme is presented. Design considerations such as prediction rate, silicon area and latency are examined and the field-based scheme is found to outperform existing prediction schemes in terms of prediction hit rate while requiring a similar amount of area as traditional schemes. In all cases the results presented in this chapter where obtained by

the author via simulations based on either the SimpleScalar/ARM or SimNP simulators and using the network traces used in previous chapters.

6.2 Performance Evaluation of Existing Prediction Schemes

Recalling the various dynamic predictors outlined in Chapter 2, this sections presents a detailed analysis of the performance of existing branch prediction schemes when applied to network workloads[1]. In general the configurations possible with a dynamic predictor are to either expand the per-branch pattern history table (PHT) or the global history register (GHR). The most basic dynamic prediction method is the directly indexed predictor. While such an architecture is sub-optimal in terms of prediction rate it does allow the saturation point for a given application to be determined. The hit rate prediction performance for a directly indexed branch predictor is shown in Figure 6.1. While the CRC application provides almost perfect prediction rates, the remaining applications all follow a similar pattern, increasing from an initially low prediction rate for ultra-small prediction tables towards a saturation point when a large number of entries are deployed. Typically, the reason for a saturation point within branch prediction relates to the problem of branch interference within the PHT. It is clear that the small application kernel found in NP applications should minimise the effect of branch interference for large table sizes, but this small application size also reduces the variance seen in the program counter, a key aspect in randomising the PHT index. Averaging the header and payload tasks into the two categories, HPA and PPA, a performance difference between both categories can be seen. While the average PPA hit rate is almost 95% for small 64-entry predictors, it is only 85% for the HPA applications. When compared to SPEC benchmark applications [179] the most noticeable characteristic of NP applications is that they will tend to saturate at a lower point when compared to general purpose applications (SPEC benchmark applications required at least 8K entries).

The more efficient prediction architectures such as *GAp* or *PAp* require both the level 1 and level 2 dimensions to be explored. In each case the specific predictor model was

[1]The results in this section utilise *gcc-2.9.5*/glibc/Simplescalar.

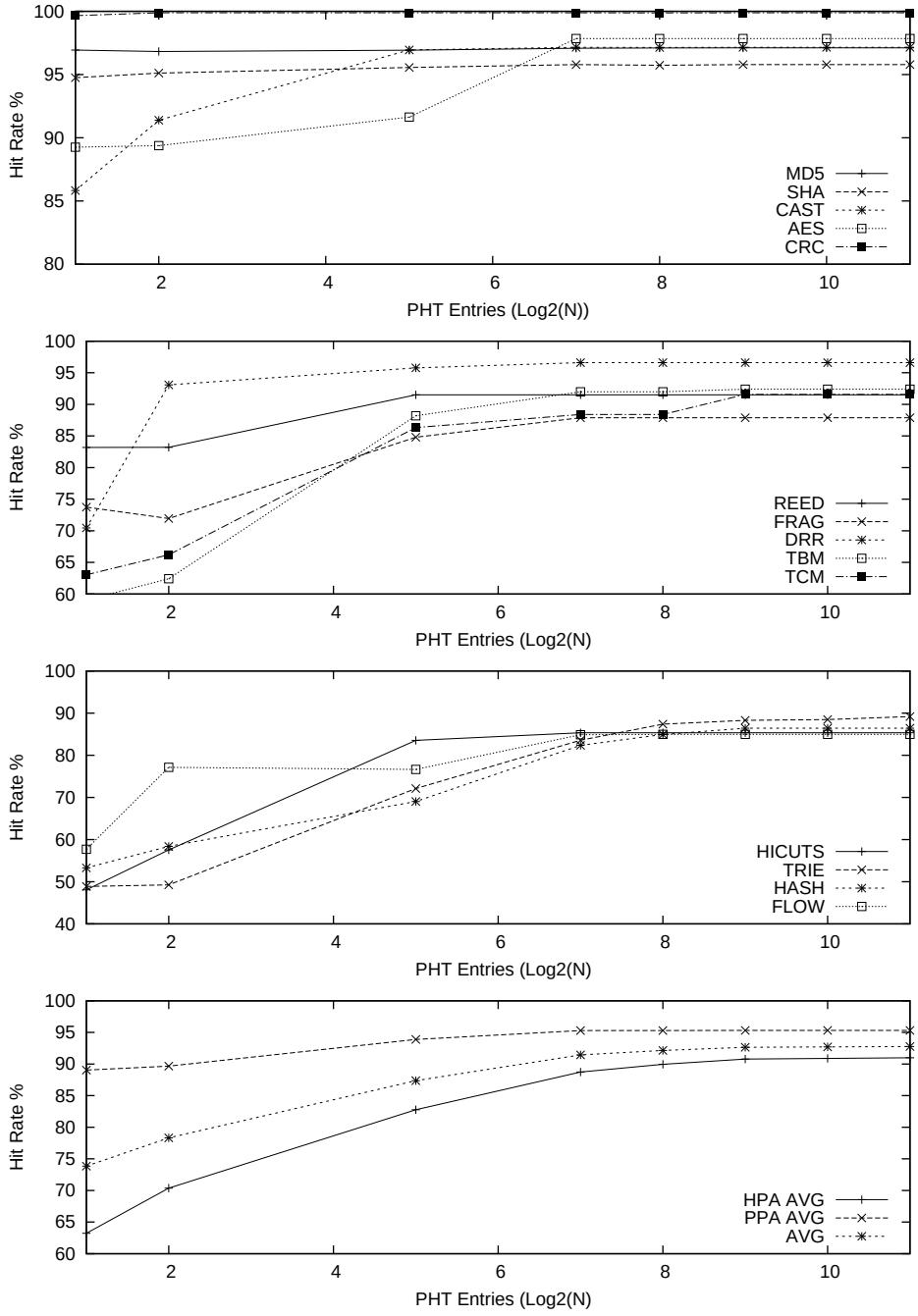


Figure 6.1: Directly Indexed Predictor Performance

implemented on the SimNP platform and the various configurations were examined via an exhaustive search. In Tables 6.1 and 6.2 the performance of various configurations of 2 level predictors is presented. The prediction hit rates are averaged across the TXS, PSC and OSU traces used previously. For comparison, a static prediction technique (assume always taken) is also presented. In the case of global address schemes, performance was found to increase in a similar fashion to both directly mapped and *gshare* based schemes,

Table 6.1: Predictor Performance For Various Global Address Schemes

Architecture	**Hit Rate %**		
	HPA	PPA	AVG
Taken	65.41	86.97	74.29
GAg-256	92.29	95.05	93.50
GAg-512	93.59	96.19	94.73
GAp-256-7	94.04	94.36	94.18
GAp-512-8	94.06	95.17	94.55

Table 6.2: Predictor Performance For Various Per Address Schemes

Architecture	**Hit Rate %**		
	HPA	PPA	AVG
PAg-(16-256)	91.24	96.88	93.71
PAg-(32-256)	91.96	96.86	94.10
PAg-(64-256)	93.38	96.85	94.90
PAg-(128-256)	93.78	96.86	95.32
PAg-(128-256)	93.78	96.86	95.32
PAp-(4-2-256)	92.58	95.63	94.05
PAp-(4-2-512)	92.23	96.07	94.15
PAp-(4-4-512)	92.19	95.82	94.00

with the *gshare* architecture consistently outperforming the *GAg* and *GAp* architectures regardless of table size. For the *GAp* predictor the PHT table size refers to the total number of PHT entries, so that a 512-8 predictor refers to a scheme in which an 8-bit GHR is concatenated with a single bit from the program counter to index one of two 256 entry PHTs.

For more complex schemes employing a Per-Address history register, the trade off is between additional per address level 1 and the number of PHT entries which can be employed. In the case of a PAg-128-256, 128 8-bit GHRs are used to index a single 256 PHT. The area requirement for such a predictor is estimated as approximately 9,200 transistors. For a *PAp* scheme the use of parallel PHTs greatly increases the area requirement, with the PAp-4-4-512 predictor requiring almost 25,000 transistors to implement. When compared to the more basic global schemes, it can be seen that neither a *PAg* nor *PAp* provide significant performance increases.

6.3 Gshare Predictor Performance

With a *gshare* type architecture identified as providing the most efficient solution to branch prediction for NP applications, a more detailed analysis of *gshare* performance is presented in this section[2], with branch interferences, utilisation and saturation each examined within an NP framework.

Similar to a directly addressed scheme, the optimum number of PHT entries must first be determined. Using the *PSC* OC-48 trace, the predictor performance for various PHT sizes was obtained from simulation (Figure 6.2). For very small table sizes (≤ 32) it can be seen that a small number of applications have very low prediction rates, with the three most complex header applications (TRIE, HYPER & STAT) all achieving a hit rate of 70% or lower.

For the payload applications only the prediction rate of the SHA algorithm increases substantially as the number of PHT entries is increased, with other PPA applications such as CRC, FRAG, CAST and MD5 all achieving a prediction rate in excess of 90% when only 16 PHTs are used. In addition to the 12 algorithms outlined in Chapter 5, four more complex combinational applications are also examined. Since it is unlikely that an application such as TCM metering, RFC classification or TRIE-based forwarding would be implemented as a standalone application, the four applications represent possible configurations. In the first case, incoming packets are marked based on the TCM algorithm before packets marked with a certain colour are classified, otherwise packets are transmitted to the egress port without any classification (TCM-RFC). The second application implements TRIE based forwarding before fragmenting large files into smaller sections (TRIE-FRAG). The next application first classifies a packet using the HYPER algorithm before determining the next hop via TRIE-based lookup (HYPER-TRIE). The final application uses the TRIE algorithm and the CRC algorithm to calculate the next hop before appending a checksum to the forwarded packet (TRIE-CRC). As can be seen in Figure 6.2, the combined applications achieve a lower hit rate when compared to the primitive

[2]The remaining results in this chapter utilise *gcc-3.4.4*/newlib/SimNP.

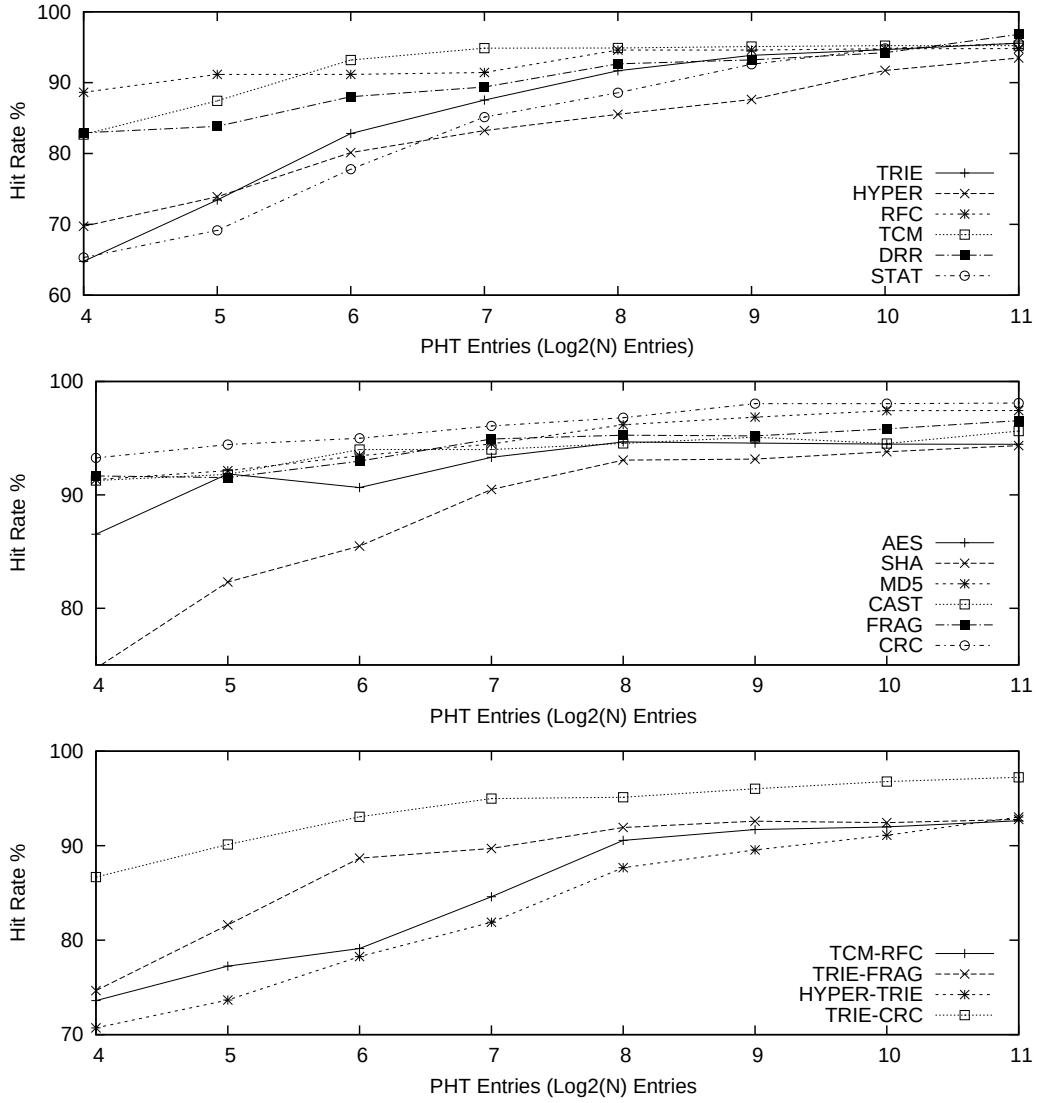


Figure 6.2: Gshare Predictor Performance for Various Network Applications

functions. This is to be expected since a higher number of conditional instructions will increase the load factor on a fixed table size, increasing the chance of branch interference between conditional instructions. As an example of this fact, the 2-K entry predictor achieves a hit rate of 96.34% and 95.72% for the FRAG and TRIE algorithms, but the combined TRIE-FRAG application achieves only 92.78% with the same predictor. With NP applications becoming increasingly complex, this point highlights that, while *gshare*-based solutions can provide good prediction rates, future developments within NP systems may make such architectures difficult to scale in terms of prediction performance.

In addition to the absolute prediction rate, in Chapter 3 two additional branch prediction metrics were outlined which allow various branch prediction schemes to be analysed.

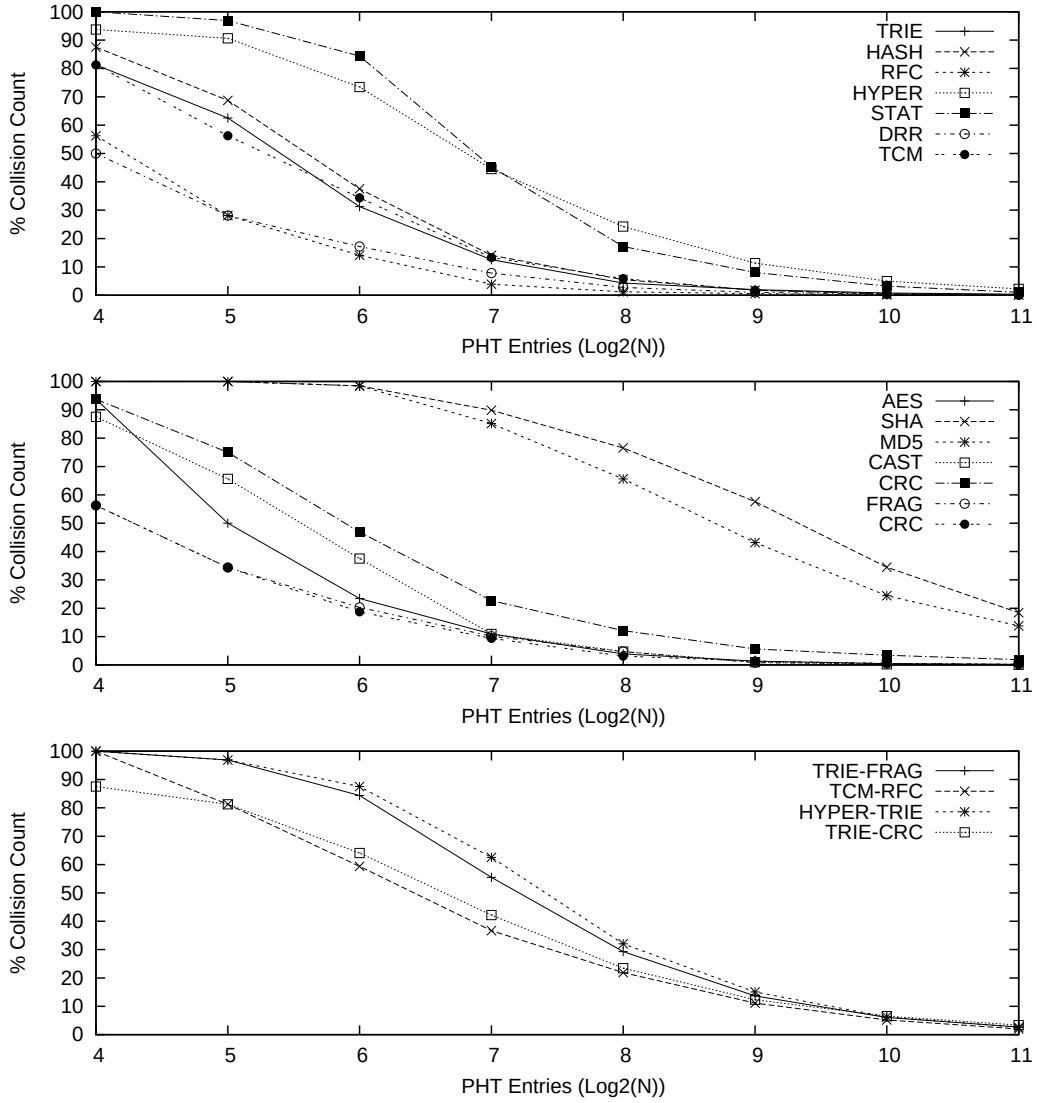


Figure 6.3: Gshare Predictor Collisions as a Percentage of PHT Size

The first mechanism is the collision rate which allows the trade-offs within the hashing scheme to be examined. By modifying the statistics maintained by SimNP during simulation it is possible to extract these two parameters. Figure 6.3 shows the collision rate for various NP applications obtained from simulation. It is clear from the figure that below 64 entries there is a high probability that any entry within the PHT table will contain some degree of branch interference, with the branch history represented in the PHT much more speculative than normal. With *gshare* employing a single global address register, it is possible for the same branch to map to multiple entries since the GHR register might not match exactly each time the branch is evaluated. In this case the branch history is distributed across multiple PHT entries, causing interference across multiple branch

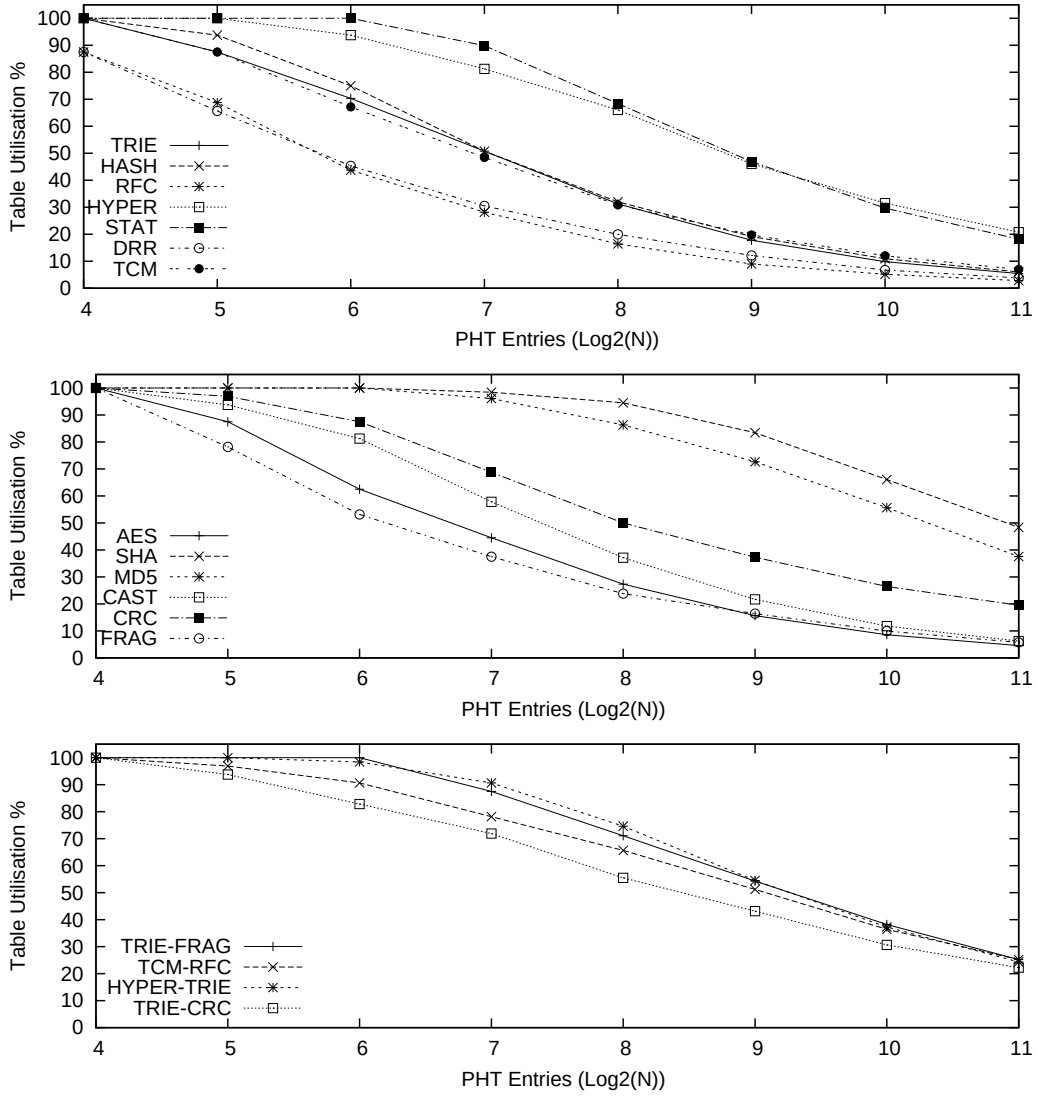


Figure 6.4: Gshare Predictor Table Utilisation

instructions. Such situations help to explain why certain header applications, such as HYPER and STAT, perform significantly worse than other applications. For the HYPER algorithm, with a maximum of 353 branch instructions per packet, a 256-entry PHT has 58 colliding entries, representing 24% of the table size. Furthermore, for all NP applications, increasing the table size to 512 or greater does not reduce the absolute number of collisions significantly. Analysis of the TRIE algorithm found 11 collisions on a 256-entry PHT, while a 2048 entry PHT only reduces the number of collisions to 7. The collision rate does not directly infer prediction performance since one of the colliding branches may only be called sparsely, or may be predicted correctly simply because the other branches at this location have evaluated in the same direction (positive interference).

The second metric which allows a predictors efficiency to be analysed is the table utilisation of a given architecture and application. In Figure 6.4 the table utilisation for each NP application was obtained from single PE simulations. In each case, with the exception of the two authentication algorithms (MD5 & SHA), no NP application achieves a table utilisation in excess of 25% for large predictor sizes (2K-Entry). In the case of common applications such as TRIE, HYPER or STAT the utilisation rate falls from approximately 100% for small 64 entry predictors to only 5.4%, 20.75% and 18.1% respectively for 2048-Entry predictors. For the large applications, the utilisation rate is in excess of 70% up to 128 entries, with an average utilisation of only 25% for the 2K-Entry predictor.

6.4 Performance Limitations of Dynamic Predictors

As was previously outlined, when compared to general purpose processing, NP applications occupy smaller footprints and therefore require a smaller number of pattern history entries. Similar to prediction rates obtained on GPP systems, above a certain size, additional table entries provide no increase in the hit rate. Expanding on this analysis, a detailed examination of the limitations with pattern prediction schemes is now presented, with each application examined in order to determine which variable(s) within the target NP application determines predictor performance and whether the average prediction rate remains the same regardless of the packet trace.

6.4.1 Payload Applications

With dynamic predictors well suited to predicting branch directions within loops, NP data almost always provides enough iterations to ensure predictor saturation. With the number of iterations defined by the input packet size and the application block size, it is possible to infer the relationship between the number of loop iterations (packet length) and the prediction rate. For an n byte packet processed in sections of p bytes, $\frac{n}{p}$ iterations of the control loop are executed. Assuming the control branch is mapped to a 2-bit dynamic counter initialised as weakly-taken, it is clear that the counter msb will correctly predict

iterations $1, 2....(\frac{n}{p} - 1)$. The prediction rate for branch *x* is therefore:

$$HR = \frac{n - p}{n} \tag{6.1}$$

Assuming the application is comprised of a number of nested loop functions, it can be seen that maximising *n* should allow increased performance, provided the number of nested control branches is higher than other 'flow' control statements. In Figure 6.5 the prediction rate for six payload applications is shown as the packet length is increased. In all cases, 1000 *n* byte packets were processed using a 64-Entry and 256-Entry *gshare* predictor. While a 64-Entry architecture provides only 91.72% and 91.89% hit rates for 40 byte packets (AES & CAST), increasing the average packet size to 200 bytes increases the prediction rates to 95.36% and 94.35% respectively. Since branch interference remains a significant problem with a small table, the increase in the prediction rate can be determined as being due to the branch instruction associated with the processing loop mapping to a saturated counter, correctly predicting all branches except the final iteration. The sheer volume of branch instructions therefore masks mispredictions elsewhere in the packet processing flow. For hashing algorithms, the packet manipulation routines are well suited to word aligned boundaries, creating a saw tooth edge (The hash key is padded between the header and payload as per IPsec Authentication Header Protocol). For the fragmentation algorithm, a destructive interference initialises the branches at a very low hit rate ($\sim$ 80%), before stabilising once the average byte count is above the fragmenting threshold.

Since it would be inefficient to pad every packet less that 100 bytes out to a minimum n_{min} bytes, it would be more useful to incorporate this information in other ways. For example, the packet fragmentation threshold could be set at a level to ensure fragments are above n_{min}. For those payload applications which have no dependency between blocks such as AES and CAST, performance increments could be achieved by grouping small packets together at encryption time, with the PE configured to encrypt a given data block before re-segmenting the encrypted buffer into their respective packets.

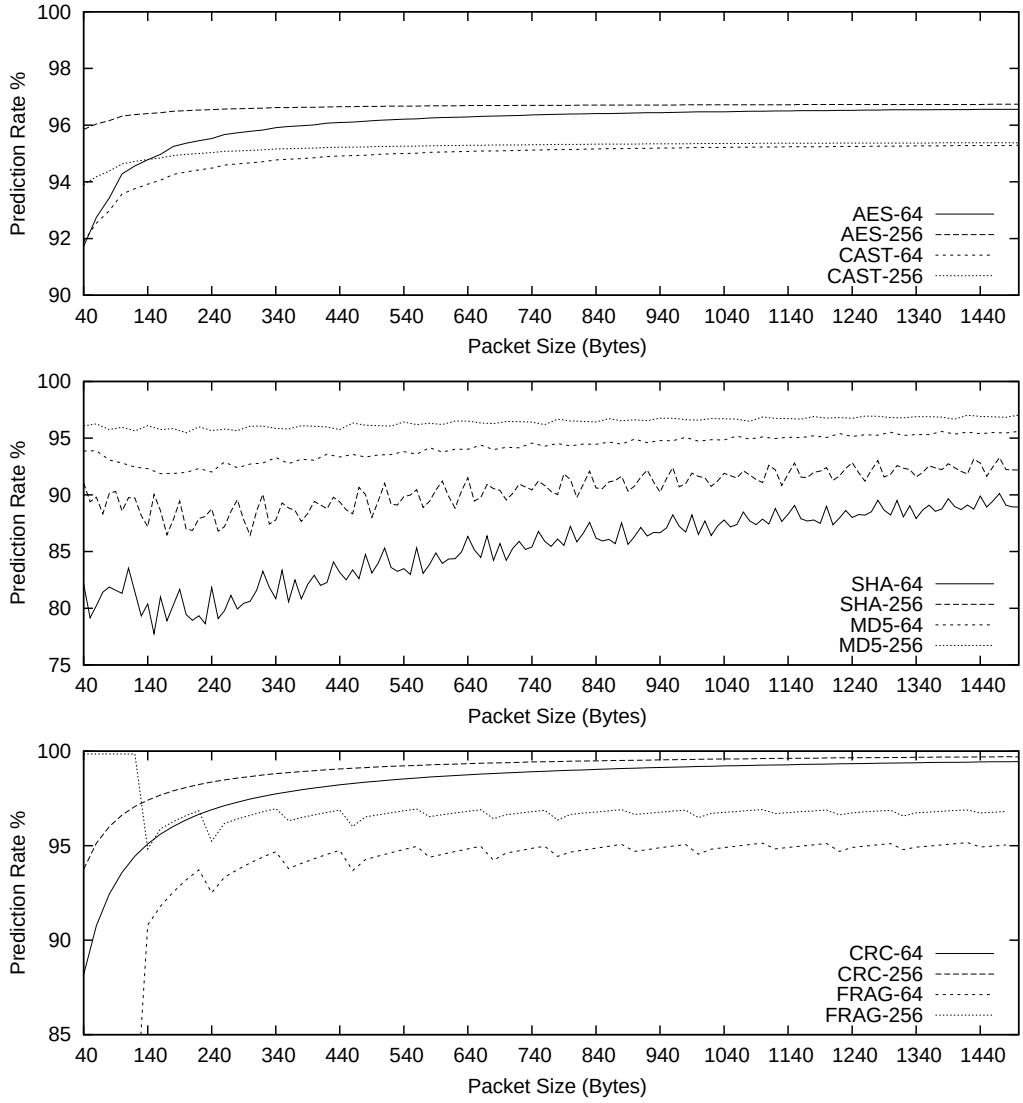


Figure 6.5: Payload Application Prediction Vs Packet Length

6.4.2 Header Applications

For header applications there are a number of possible variables which can directly effect prediction performance. While some of these variables can be safely ignored, e.g. the percentage of corrupted packets traversing any IP network, the more important question is how prediction rates change over time.

6.4.2.1 Forwarding Applications

For IP forwarding applications, such as TRIE and HASH, the forwarding table represents the most dynamic source within the application. Changing as new networks and routes are added, it is difficult to isolate the routing table from the underlying network topography.

In the case of network simulations, the lack of availability of real-world network traces requires shared repositories such as NLANR [164] to be used. For the SimNP simulations, the anonymised addresses are replaced with addresses derived from the destination addresses referenced in common routing tables. Regardless of the forwarding structure utilised, the prediction rate is more likely to be changed by the underlying data rather than the absolute number of entries. To examine this, routing entries from the MAEWEST and AT&T East Canada routing table are parsed to form a new synthetic routing table. The results in Table 6.3 demonstrate that although prediction rate will change as the routing table is altered, the difference in performance is relatively small, 1.45% for TRIE-based forwarding and 1.03% for HASH based forwarding.

Table 6.3: Gshare Prediction Hit Rate For TRIE and HASH Forwarding

TRIE		**HASH**	
Routing Entries	**Hit Rate %**	**Routing Entries**	**Hit Rate %**
75,000	93.72	5,000	96.03
102,000	92.33	10,000	95.66
119,000	92.96	15,000	95.19
141,000	92.27	20,000	96.22

6.4.2.2 Classification Applications

For packet classification algorithms, the prediction rate is determined by both the ruleset entries and the structure used to represent the ruleset. In the case of RFC, the data structure requires no conditional operations during the rule lookup, examining the memory structure in the same fashion regardless of the underlying data. In Table 6.4, the performance of a 256-Entry *gshare* predictor is outlined as the number of classifier rules stored is increased. For this simulation a 1000 rule classbench [176] defined ruleset was used. Similar to the forwarding algorithms, the hit rate does not appear to either increase or decrease with the provision of additional rules. The prediction rate can change by up to 2.58% between one classification set and another, highlighting some of the variance in dynamic predictor performance.

Table 6.4: Prediction Hit Rate For Hypercuts Classification

Rule Entries	Hit Rate %
250	86.52
500	88.98
750	86.44
1000	86.40

6.4.2.3 Metering & Queueing Applications

The final application types examined in detail are the metering and queueing applications such as Three Colour Metering (TCM) or Deficit Round Robin (DRR). As described previously, metering algorithms such as either Single-Rate TCM, Two-Rate TCM, Leaky Bucket or Token Bucket typically operate by regulating the packet output in order to match a bucket which is configured to fill with tokens at a given *fill rate*. In the case of TCM, two buckets are used during normal operation; command and peak buckets. Both buckets are configured to fill at different rates, allowing a greater degree of granularity to be employed during metering. A sample configuration might be for the command bucket to be used to detect a large number of packets arriving within a short amount of time, while the peak bucket can be used to detect when a high number of large packets arrive within a short amount of time. In this case, packets falling into the command bucket are marked green, packets falling into the peak bucket are marked yellow, otherwise (low network load) packets are marked red. To examine predictor performance for various network conditions, the same 100,000 packets from the OC-48 trace are metered for various configurations of the fill rates. The results are summarised in Table 6.5, with the prediction rate varying between 98.1% for periods of time where network load is relatively low ($\gg Red$) and 93.13% when a high proportion of the packets exceed the peak and command fill rates.

The final application examined is the deficit round robin queueing algorithm. Similar to other queueing systems, there are three variables within the algorithm which can be identified as possibly altering prediction rates; the number of unbalanced input queues (N_{ip}), the number of output queues, (N_{op}), and the quantum associated with each round

Table 6.5: Prediction Hit Rate For TCM Metering

Red %	Green %	Yellow %	Hit Rate %
0.20	4.34	95.46	98.1
22.54	4.34	73.13	97.71
50.41	49.59	0	95.83
20.31	54.39	25.30	94.7
22.54	65.51	11.95	94.98
50.42	40.20	9.38	94.34
44.63	35.45	19.91	93.13

(Q_{rr}). The quantum within the DRR algorithm refers to how many bytes are moved from the input to the output during each round of the algorithm, so that, for example, if the current packet at input is 500 bytes long, the current queue quantum is 300 and the quantum added per round is 100, the packet must wait 2 iterations before being moved to the balanced output queue. Using the OC-12 packet trace, the prediction rate for a 256-Entry gshare predictor is shown in Figure 6.6. In Figure 6.6(A) the prediction rate is shown for a varying number of input and output queues. As can be seen, in both cases the prediction rate increases as the number of queues is balanced before falling almost 3% as the number of configured queues (either input or output) exceeds the number of fixed queues (either input (N_{ip}) or output (N_{op}). While it is clear that the relationship between the number of input/output queues will affect the prediction rate, the quantum size has no definitive relationship to the hit rate. For configurations involving a large number of queues, the prediction rate changes by approximately 1% as the quantum is increased from 100 to 1200. A quantum of 1200 would allow nearly all packets through within a single iteration, minimising the ability of the algorithm to balance the output queues.

6.4.3 Summary of Predictor Performance

While Section 6.3 examined the performance of a *gshare* under fixed conditions, the analysis outlined above attempts to quantify branch prediction performance as the underlying network conditions are varied. In the case of payload applications, maximising the packet length provides one method of improving PE performance. For all payload applications

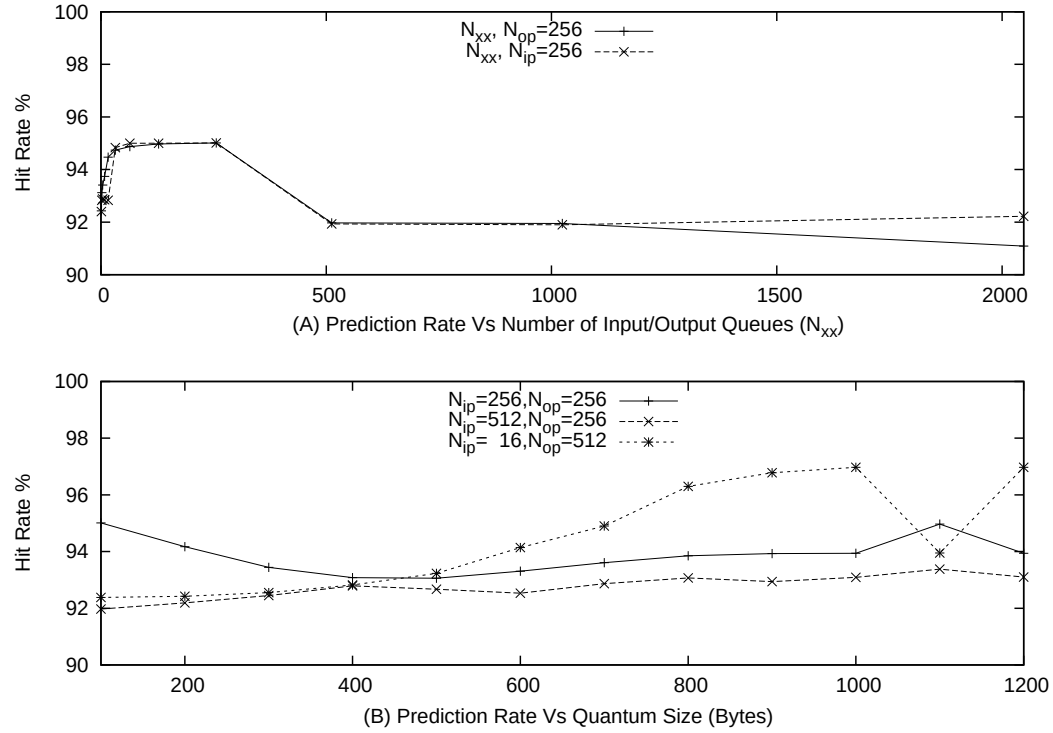


Figure 6.6: Prediction Rate For DRR Algorithm

except fragmentation, is was seen that a small 64-Entry predictor can provide similar performance to a 256-Entry predictor provided the packet length is long enough to mask some of the mispredictions. To maximise prediction rates, two methods are available which can easily incorporate this information at little cost. Firstly, the fragmentation size, either at host or router level, could be set in such a manner so as to ensure any fragmented packets are above a threshold determined by expected router performance. Secondly, regardless of flow state, it may be possible to concatenate small packets into a larger packet if the target application is found to be data block independent. However, since both fragmentation and flow-level application behaviour are topics more likely to be changed at a host level, neither of these topics is explored in detail (the IPv6 standard assumes packet fragmentation is handled by end nodes only). For header applications it was seen that both IP forwarding and packet classification are relatively insensitive to changes in the underlying control structures. For applications which store information in a trie structure, the *shape* of the trie will have more influence on prediction rates than the absolute number of entries. Larger networks, as well as the spread of IP connected systems to nations such as China and India, necessitate both wider and deeper trie structures, with IPv6 providing

a large amount of addressable users.

Queueing and metering applications are found to exhibit a large degree of variance as underlying parameters and network conditions are altered. In the case of the TCM metering algorithm, changes in how packets are metered and therefore coloured significantly alter the prediction rate, while for a software implementation of the DRR algorithm, the prediction rate is likely to be determined by the number of input and output queues employed, with modern routers employing thousands of queues. With these results in mind, the challenge for achieving high PE utilisation is how additional prediction performance can be achieved when the dynamic branch behaviour outlined in this section is taken into account.

6.5 Utilising Packet Flow Information during Branch Prediction

6.5.1 Flow Information For Payload Applications

For payload applications, the *majority* of the branch history represents branch operations corresponding to the control loop operating on the packet payload. Consider the pseudo code for a packet encryption and encapsulation application outlined in Listing 6.1. Once the packet has been fetched, the header and payload are encrypted on a fixed per-block basis. The *encrypt_packet* function encrypts the next *block_size* number of bytes starting from *pkt_ptr* and attaches the result to *enc_pkt*. The while-loop continues until every byte of the packet (including the header) has been fetched and encrypted. Then the encrypted packet is encapsulated with a new IP packet header for transmission. Generalising this framework to all payload applications it can be seen that to attach a digital signature or checksum to the outgoing packet, the function *encrypt_packet* would be changed while the encapsulation routine would also be altered to reflect the new requirements. However, the overall programming design would remain the same.

With the branch history growing linearly, it can be seen that while the branch history

should represent a summary of previous branch history, it is likely to be only the history of the immediately previous packet. Examining the sample code outlined below, it is clear that if the packets *i* and *i+1* are of equal length, the execution path traversed should be identical (assuming no processor/packet exceptions). Analysing the instruction paths for payload applications it is found that for applications such as AES, CAST, CRC or FRAG, the execution path grows linearly.

```
packet_encapsulate(char* pkt_ptr) {
    struct ip iphdr, new_iphdr;
    char* enc_pkt;
    iphdr = fetch_header(pkt_ptr);
    while(iphdr.ip_len) {
        encrypt_packet(pkt_ptr, enc_pkt, block_size);
        iphdr.ip_len -= block_size;
        pkt_ptr += block_size;
    }
    new_iphdr = new_header(iphdr, enc_pkt);
    pkt_ptr = encap_packet(new_iphdr, enc_pkt);
    transmit(pkt_ptr);
}
```

Listing 6.1: Pseudo Code for Encryption and Encapsulation

In Figure 6.7 the branch history string for the AES application is shown. The branch string represents a bit sequence formed by the evaluation of all branches during packet processing, i.e. a '1' represents a taken branch while a '0' represents a not-taken branch. In the figure, the hexadecimal value *105FF001* describes 32 branch operations, were the first three branches where not taken, the forth branch was taken, followed by 4 not taken branches, etc. For the AES algorithm, all valid packets have the exact same execution history for the first 121 branch operations, while those packets greater than 48 bytes continue from this point on. For other payload applications a similar behaviour can also be seen, with the exception of the hashing algorithms (MD5 and SHA-1) which have a number of paths at block size boundaries but identical bit sequences between these block size

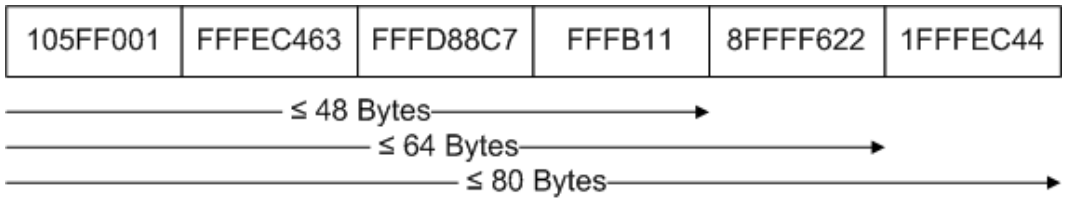


Figure 6.7: Execution Path For AES Algorithm

boundaries.

Since the PHT branch history represents the sum of previous packet lengths, one method of improving branch prediction performance would be to section the branch history based on the header field which differentiated the execution path for packets of varying length. Recalling the dynamic branch behaviour analysis presented in Chapter 5 it is obvious that it would not be feasible to completely retain the branch history for each possible packet length, requiring too many separate entries and too much cache history. However, an analysis of network traces finds that certain packet lengths represent the majority of IP traffic (Table 6.6). For the 1.7 million packet PSC trace, 8 individual packet lengths comprise over 76% of the total packet trace. By incorporating this information at run time, improvements in the branch prediction rate should be possible. It should be

Table 6.6: Detailed Packet Distribution For OC-48 Trace

Packet Length	**Percentage of Trace**
40	17.10
52	15.54
1420	7.74
1500	30.10
Total	70.48

noted that more general packet distribution studies in [180], [181] and [182] found that in addition to high proportion of small ($\leq$ 100) and large ($\geq$ 1000) packets, another distribution peak can be seen at packet lengths of approximately 500 bytes. However, none of the traces used in this study had such a peak, but it is expected that such a distribution would further increase the performance of a flow-based predictor scheme since it further reduces the variance of packet lengths which must be indexed.

6.5.2 Flow Information For Header Applications

While payload applications typically follow a loop framework, header based applications are more commonly implemented as a number of *if-else* statements. An example of this is shown in the IP forwarding pseudo code outlined in Listing 6.2. In this example, a packet pointer *pkt_ptr* is passed to the function *packet_forward()* for processing. Once the whole IP packet header *iphdr* has been fetched and verified, the next hop address is determined based on the destination IP address. If the header or the next hop is not valid, the packet is dropped. Otherwise, it is modified (e.g. decrementing the Time-To-Live field of IP packet header) and forwarded.

```
packet_forward (char* pkt_ptr)
{
  struct ip iphdr;
  int next_hop;
  iphdr = fetch_header(pkt_ptr);
  if(verify_header(iphdr) == TRUE) {
     next_hop = find_next_hop(iphdr.dst_address);
     if(next_hop == PROBLEM)
        drop(pkt_ptr);
     else {
        modify(iphdr);
        transmit(pkt_ptr);
     }
  } else
     drop(pkt_ptr);
}
```

Listing 6.2: Pseudo code for IP Forwarding

Assuming the vast majority of packets pass verification, the execution path can be seen to be most affected by the destination address. Similar to predicting branch direction by segmenting branch history by sizes, the branch direction for an application such as IP

forwarding can be predicted by assuming two packets with the same destination address will follow the same execution path.

Applying the same principal to other header applications it is possible to extract packet fields which can be used to identify the most likely execution path. For packet classification, statistics or NAT applications, packets of the same flow will have the same execution path, with flow identification possible at a 2, 3 or 5 tuple level. For queueing and metering algorithms it is not possible to deduce a single variable by which future branch instructions can be predicted based on past packet history. For the metering algorithms, the inter-arrival time will determine the number of tokens added to the bucket during the inter-arrival period and therefore influence the prediction decision(s). For queuing algorithms such as DRR, the packet length does provide a hint as to the execution path which might be taken but, with no reference to the quantum (or more specifically the quantum per round), using the packet length to predict how a packet will be processed will be wrong a certain proportion of times.

It is clear that for payload applications it is possible to determine future execution paths based on how previous packets of the same length have been processed. Similarly, for packet forwarding applications it is possible to predict future branch decisions based on how previous packets routed to the same destination addresses were processed. These header fields (Search Keys) can be further expanded to include flow based applications such as RFC, HYPER or STAT, whereas for metering and queueing algorithms, a search key based on packet length represents a method of indexing one possible execution path. A summary of possible IP based search keys by which application branch history can be sectioned is presented in Table 6.7.

6.5.3 Indexing Branch History

Given that the Search Keys (*SK*) outlined in the previous section can be used to partition branch history on a per-packet basis, the question is how this information can be used to improve branch prediction rates within a PE. Consider a PE which operates on a run-to-completion basis, where packets are either requested from a central arbitrator or popped

Table 6.7: Flow-Index Search Key Extracted From Packet Header

Architecture	Search Key
TRIE	Source IP + Destination IP
HASH	Source IP + Destination IP
HYPER	Source IP + Destination IP + Protocol
RFC	Source IP + Destination IP + Protocol
STAT	Source IP + Destination IP + Protocol
DRR	Length
TCM	Length
AES	Length
SHA	Length
CRC	Length
FRAG	Length + Offset

off the front of a queue mechanism. Once packet *i* has been allocated to the waiting PE, the packet can be processed. The Branch History (*BH*) of packet *i* can be defined as the concatenation of the *m* conditional branch operations encountered during the packet processing stage

$$BH_i = c_0.c_1.c_2...c_{m-1} \tag{6.2}$$

Seen as single branch history would not provide much additional information, *n* previous packet histories must be stored, creating an $n * m$ branch String Table (*ST*).

$$ST = \sum^{m} -1_{i=0} = BH_0 \cup BH_1 \cup BH_2 \cup ...BH_n \tag{6.3}$$

Future use of BH_i to predict branch behaviour of packet *k*, both of which have matching search keys ($SK_k =\in ST$), depends on the PE being able to index the string table based on a search key instead of an absolute address. A number of methods of performing this function are possible. Firstly, the search keys can be stored in a tag memory associated with each *m*-bit string. Given a search key, a hardware mechanism to linearly search all *n* elements can be performed to identify the correct location. Parallelisation of this search function is possible since multiple locations could be examined at the same time. Assuming the indexing function must be complete in a single clock cycle, *n* indexed

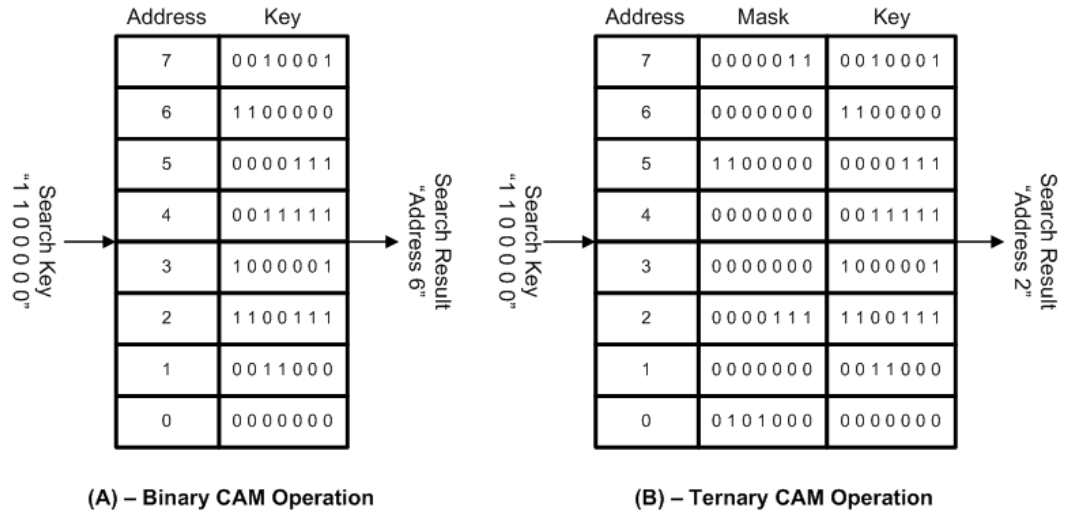


Figure 6.8: Binary/Ternary CAM Operation

search keys, each of which is *log2(n)* bits long would require $n * log2(p)$ SRAM bits and *n log2(n)*-bit binary comparators (XOR operations).

A second method would be to utilise the search key in a hash algorithm to create a hash-indexed structure. As with all hash structures, the lookup time to find the index address would be *O(1)*. Replacing the parallel subtractors required for a linear search with a single combinational hashing block, the primary difficulty with a hash structure is that for small values of *n*, hash collisions would present a serious limitation to predictor performance. For example, if the predictor was required to retain the branch history of the past 256 (*N=8)* packets, the birthday paradox implies a hash collision for every $2^{\frac{N}{2}}$ different inputs. As with the hash schemes employed by branch prediction, the need to heavily reduce the mapped space implies a high table load factor, with the corresponding collision rate.

Another possible method by which the string table can be indexed is to use a CAM structure [183]. Unlike normal SRAM which accesses data based on an address presented to the SRAM logic, content addressable memory operates by examining all locations with a search key before returning the addresses of the locations where the content matches the search key (if any). Available in either Binary or Ternary format, the difference is outlined in Figure 6.8. For a binary system, an exact match between the search key and the stored values results in the address being returned, while in a ternary CAM certain bits

within the search structure can be masked off as 'don't care' states via the mask register. In the example shown, two locations within the TCAM match the search key, with the lowest location being returned by precedent. For flow-indexed branch prediction, a CAM structure allows a binding between the search keys and history registers to be created with no chance of collision across search keys.

6.5.4 Field-Based Branch Predictor

With each location of an *N* entry CAM mapping to an address offset within an *N*M*-bit string table, it is possible to create a full-associative Branch String Table in which the index can be easily accessed via a CAM search. While it is clear that the branch history can be used to guide future predictions of packets matching the search key, there is no history available for the first packet of a matching key-flow[3]. To solve this training time, a Fall-Back (*FB*) predictor must be present while the branch history is being collated. Furthermore, since the memory allocated to each search key may be less than the number of branch operations evaluated during processing, the fall-back predictor must also provide prediction if the stored branch history in the *ST* has been exhausted. Assuming hit rate of the string table prediction and fall-back predictor is HR_{ST} and HR_{FB}, the overall prediction rate for a packet with search key SK_k and a branch history of length *M* can be estimated as:

$$PR_k = \begin{cases} HR_{ST} & \text{if } SK_k \in ST \;\&\; M \leq M_{threshold} \\ HR_{FB} & \text{if } SK_k \notin ST \\ HR_{FB} & \text{if } M \geq M_{threshold} \\ HR_{FB} & \text{if } T_{miss} \geq T_{threshold} \end{cases} \tag{6.4}$$

where $M_{threshold}$ and $T_{threshold}$ are the two configurable threshold values used to determine predictor behaviour when the current branch history length exceeds the stored branch history and $T_{threshold}$ is used to allow fall-back prediction when the string history does not accurately represent the current packet execution path. Once the number

[3] A key-flow is defined as those packets which having matching search keys

of mispredicts exceeds the threshold, the branch history is assumed to no longer match the packet execution and so control is passed to the fall-back predictor. In addition to predictor behaviour during normal operation, another issue which must be resolved is the methodology employed when allocating a CAM entry to new search keys. For header applications such as IP forwarding or packet classification it is assumed that a search key will time out as the underlying network flow terminates. To ensure fairness, a new search key is allocated to the next CAM location, regardless of the current state or how often the CAM location is accessed. As such, the CAM allocation algorithm can be simplified to a round robin mechanism and is implementable via a simple modulo counter. While the branch string width (M) relates to the precision applied to each packet matching a given search key, the number of CAM entries (N) regulates the number of active flows retained at any point in time, or the amount of time a specific branch history remains available to the predictor. For certain payload applications, a semi-static branch prediction methodology might be to pre-compute and allocate a certain number of the CAM entries (and branch history strings) to those search keys which are highly probable to occur again, e.g. 40, 64 and 1500 byte packets.

A block diagram outlining a field-based predictor scheme is shown in Figure 6.9. Using a simple clocking scheme, Figure 6.10 presents a timing diagram for such a predictor scheme. In the first instance, a new packet allocated to the PE triggers the *bNewPacket* signal which allows the internal address logic to be reset. While the address logic is being reset, the new packet signal can be used to latch the search key into the predictor logic. On the next clock cycle the CAM unit either returns a match or signals that the search key must be allocated space within both the CAM and string table. In the timing diagram it is assumed that the search key matches contents found in CAM address *ST Address*. Using the *bMatch* signal the output multiplexer is to configured to select the least significant bit of the T-bit O/P shift register. At the same time as the multiplexer is being set, the *ST Address* value is used by ST address logic to calculate the index address in the string table. To save on having to access the string table for each branch prediction, the T-bit input and output registers allow the string table to be accessed in T-bit blocks, fetching

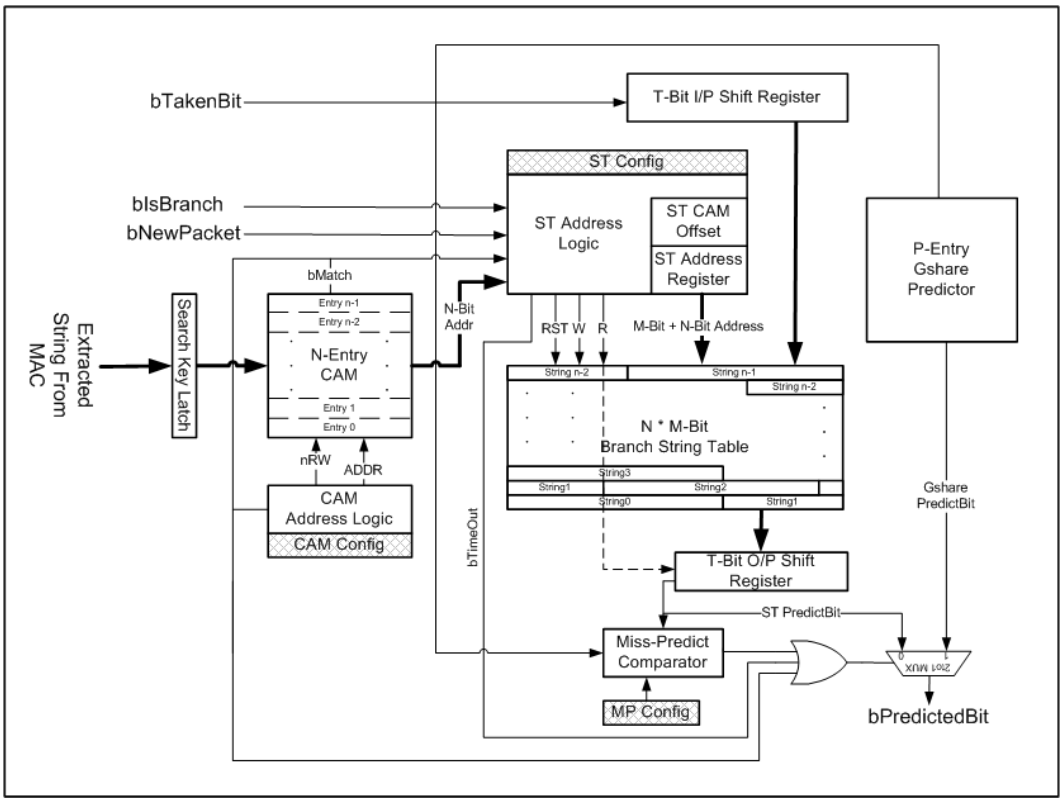


Figure 6.9: Block Diagram of Field-Based Predictor

the next T branch operations during read operations and writing T bits when updating the string table history. At the cost of an additional register, such a configuration allows the critical path of the design to be reduced to a signal shift register during normal read-mode predictions, with the *bIsBranch* signal shifting the contents of the O/P register by one place. Since space within the CAM logic is allocated on a fair basis, the address logic used within both the CAM and string table logic can be implemented as simple modulo counters. As can be seen, the prediction architecture can be expanded in three directions, through additional CAM entries, a longer branch history per entry or by expanding the number of PHT entries within the fall-back predictor. The CAM width is configured as 32-bits wide, requiring certain values to be compressed before concatenation into a search string, while other small values such as the 16-bit packet length is padded out with leading zeros.

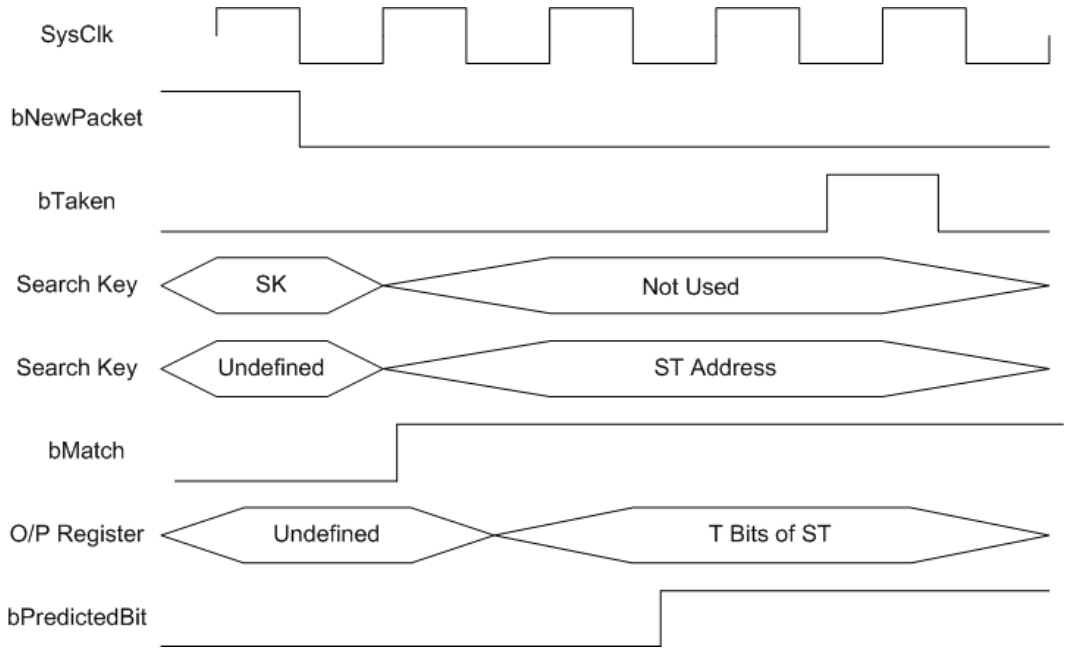


Figure 6.10: Example Timing Diagram for Field-Based Predictor

6.6 Performance Evaluation of a Proposed Predictor

Utilising the architecture outlined in the previous section, an analysis of the performance is presented in this section. A software model for the field-based predictor was implemented within the SimNP simulator, allowing the proposed architecture to be compared to the results presented previously without any change in other system parameters. Firstly, the latency and area requirement of a field-based predictor is examined to ensure the design constraints are realistic. Secondly, the utilisation of a field-based scheme is presented since the branch history must be used significantly more than the fallback predictor to justify such an architecture. Thirdly, an examination of impact of both additional CAM entries, longer branch history width and a larger fallback predictor is presented for both individual and combinational applications.

6.6.1 Latency of Field-Based Branch Predictor

As was shown in the Figure 6.10, there is considerable scope for hiding the latency associated with CAM lookup timing in a manner such that the delay is transparent to the overall prediction function. Considering it is unlikely that the CAM lookup and string table indexing could occur in a single clock cycle a more obvious solution would be to

trigger the *bNewPacket* signal as early as possible, processing the lookup function while the packet is still being allocated to the PE.

A sample configuration might be a situation where the PE requests a packet from a centralised queue arbitrator. On receipt of a packet request, the arbitration unit returns the search key, packet address and packet length to the requesting PE. Provided the very first instruction in the processing stream is not a branch instruction, it is therefore possible to process the setup functions before packet processing occurs. The CAM read time is τ_{cam}, the string table read time is τ_{string} and the fall-back predictor read time is τ_{gshare}. Generally speaking, the delay associated with CAM read operations is heavily dependent on factors such as the number of entries to be searched and the speed at which one CAM location can be clocked. In CMOS, TCAM architectures can be operated at very high frequencies (3nS in 0.18um technology [184]). For large external TCAMs, a typical implementation might be to clock the device at this speed by processing only a section of the TCAM entries during each clock cycle, creating a multi-cycle lookup which must be pipelined. Since the CAM structure employed in the proposed branch predictor is limited to only a small number of entries the CAM lookup operation can be performed fast enough in a single cycle. During normal execution, the branch predictor must either trigger the O/P register or wait for the *gshare* predictor to determine a prediction direction. The time delay for each prediction bit is therefore

$$\tau_{predict} = \begin{cases} \tau_{reg} & \text{if k} \in \text{ST} \\ \tau_{gshare} & \text{all other conditions} \end{cases} \tag{6.5}$$

Similar to the update function employed within traditional dynamic predictors, control operations associated with updating the O/P register, updating the misprediction conditions, checking the string length and writing the prediction bit back to the predictor can be arranged in such a fashion so that they occur between branch operations.

6.6.2 Chip Area of Field-Based Branch Predictor

Similar to previous prediction architectures examined, for a field-Based solution to be viable, the transistor cost must be small enough when compared to the overall PE area. Using either a 6 or 9 Transistor CAM cell, ([184] and [183]), the area of an *N/M/P* predictor can be estimated as[4]:

$$A_{field} = (6 * N * CAM_{width}) + (6 * N * M) + (6 * 2 * P) \tag{6.6}$$

For a 32-bit wide CAM structure, the area of a 32/128/128 field-based predictor would be 32,256 transistors, comparable to the 24,000 transistors required for the 2K entry *gshare* predictor. Recalling that the utilisation of a *gshare*-based scheme quickly falls off above 256 entries, the original transistor budget of 25% of PE area allows various configurations to be examined. When compared to 32/128/128, another configuration might favour a longer string history per search key with a smaller amount of searchable elements. For example, a 16/512/256 field-based predictor allows a large amount of branch history to be retained, increasing prediction precision since matching packets have almost perfect prediction, providing the history length is shorter than the threshold ($M_{threshold}$). Such a configuration would require ∼54,000 transistors to implement and would provide enough branch history to encompass all header applications and those small packet lengths processed by a payload application.

6.6.3 Utilisation of Field-Based Branch Predictor

In addition to meeting the area and latency requirements, a field-based prediction must also be utilised enough to justify the additional silicon.

As can be seen from Table 6.7, four types of search strings can be used to cover all 11 applications. In Table 6.8 the utilisation rate for 1,000,000 packets of the PSC trace is given. With only 16 CAM entries, the CAM logic is used for over 92% of packets processed during IP forwarding, with the proportion rising to 97% when *n=128*. For

[4]The area associated with the address logic, memory decoders and latches is negligible when compared to the area of either the CAM logic or the branch string table

Table 6.8: Percentage of Packets Predicted Via Field-Based Scheme (PSC Trace)

Search Key	N=16	N=32	N=64	N=128
TRIE	92.53	91.05	96.61	97.47
HYPER	86.23	89.38	93.77	95.33
AES	86.07	89.38	92.43	95.04
FRAG	86.07	89.38	92.43	95.04

the other applications, the utilisation rate is consistently over 86% for a 16 entry device, increasing to 95% when a 128-entry CAM (N=128) is employed.

6.6.4 Performance Evaluation

Examining NP applications it can be intuitively deduced how a particular application should behave when a field-based system is deployed. For payload applications, a high proportion of small packets will increase the prediction rate when compared to a *gshare* system since the entire branch history can be cached. For header applications it is clear that a string table length in excess of the maximum branch count will provide no additional performance benefits. For such configurations it would be more beneficial to implement a large amount of CAM entries, with the address logic capable of being configured so that it is possible to have either a fully associative relationship between the CAM table and the string table, or by disabling the top n entries of an N-entry CAM it would be possible to expand the string table history allocated to each of the N-n enabled entries. A detailed analysis of this behaviour is now presented, with the NP applications grouped by functionality.

6.6.4.1 Header Applications

The most important function required by a modern router, previous sections highlighted that while a *gshare* predictor provides better performance than other two level schemes, in general, all such prediction architectures will tend to saturate between 256 and 512 entries. In Figure 6.11 the prediction rate for the two forwarding applications is compared to two base *gshare* configurations (GS-256 and GS-512). For the Field-Based scheme (FB), the

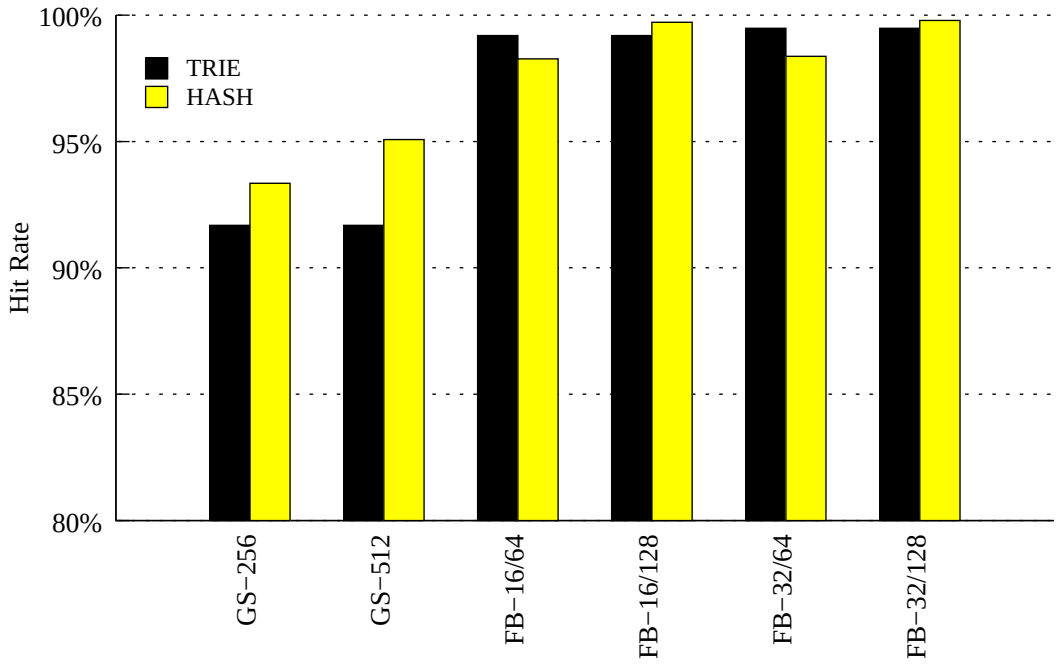


Figure 6.11: Field-Based Predictor Performance For Forwarding Applications

fall-back predictor employed is a *gshare* scheme with 128 PHT entries. The two digits within the identifier relate to the number of CAM entries and the size of string table per entry (N/M).

As can be seen in Figure 6.11, the prediction rate for both algorithms is improved using a field-based system. For the TRIE algorithm, the hit rate is increased from 92.08% for a 512 PHT *gshare* predictor to over 99.19% for field-based predictor with 16 CAM entries. The short branch history of the TRIE algorithm means that provision of a larger string table does not improve performance since the entire packet branch history can be stored within a 64-bit string table. On the other hand, additional CAM entries do provide a small degree of performance increase (99.48%). For the HASH algorithm, two aspects are of note. Firstly, the larger branch history makes the HASH algorithm more sensitive to the string table width. A change in the string table width from 64-bits to 128-bits increases the prediction rate from 98.27% to 99.42%. Secondly, since the string history would be exhausted only half way through packet processing, it is clear that the *gshare* scheme must be less well suited to traversal of a highly linked hash structure, since the first half of the application roughly translates to data retrieval. As with the TRIE algorithm, additional CAM entries provide little performance increase, although for higher speed connections

(AMP and PSC) this relationship would be more apparent.

For classification based algorithms such as RFC, HYPER or STAT, the prediction rate is shown in Figure 6.12. The STAT algorithm implements a flow-based identification system, with packet flows identified via the five tuples, and statistics maintained dynamically on a per-flow and a global basis. Both the RFC and STAT algorithm require a 64-bit history for each flow, while the HYPER algorithm requires storage for up to 353 branch operations per packet. For clarity, only the results for 256-bit string table widths are shown in Figure 6.12, although it is clear that only the HYPER algorithm requires table size greater than 64-bit and that a more optimised implementation of the HYPER algorithm could significantly reduce the amount of branch operations required per packet. While the RFC and HYPER algorithm do not modify the underlying search structure, the STAT application presents a challenge in that a mapped flow entry may be deleted if the flow terminates. For situations such as this, the cached branch history may only represent part of current conditional path, with a slightly different execution path as the flow statistics are updated. As with the forwarding applications, a 128 PHT entry *gshare* predictor is employed as the fall-back scheme. For the RFC algorithm, a 64/64 field-based predictor provides a prediction rate of 99.64%, an improvement of over 5% from the 512-entry *gshare* scheme (94.59%). For the HYPER algorithm, a 256-bit string table does not cache the entire packet history. However, performance is still increased from only 87.61% for a 512-entry *gshare* scheme to over 98.80% for a 32/256 field-based predictor. Similarly, the field-based scheme improves performance for the STAT algorithm by at least 4%, but, with a more dynamic nature, it does not reach the prediction rates of other similar classification algorithms.

Finally, metering and queueing algorithms are examined. Both applications have relatively short branch histories per packet and so can be fully cached with a 32-bit string history width. As was previously discussed, the metering algorithms such as TCM and TBM are highly speculative since the execution path in both algorithms is determined by a largely non-deterministic value, namely the packet inter-arrival time. While it would be possible to use a scaled version of inter-arrival time as a search key within the field-based

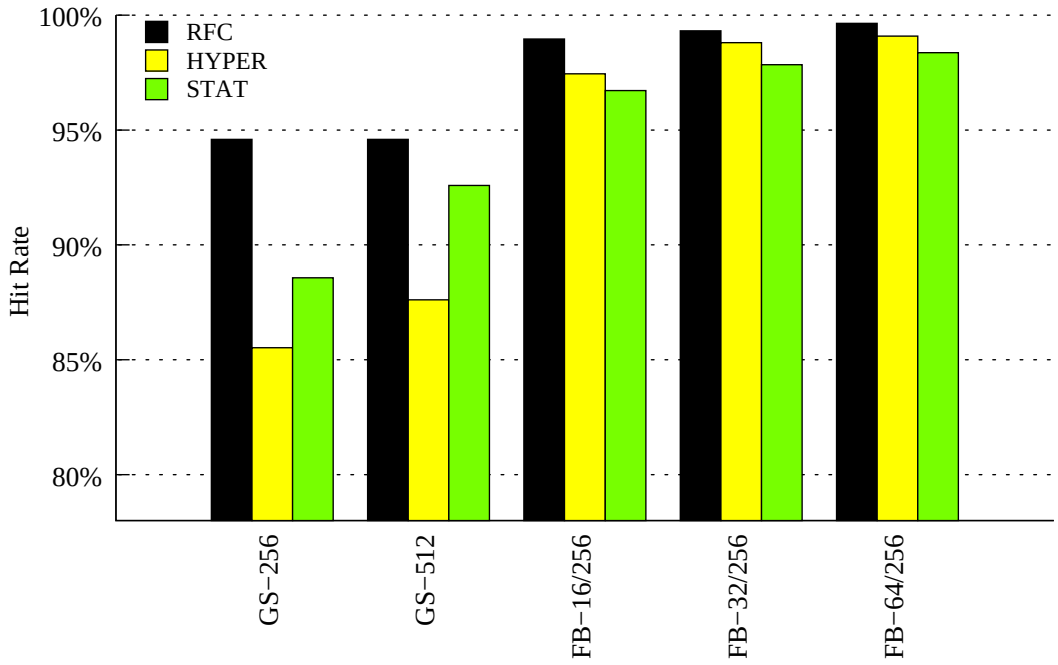


Figure 6.12: Field-Based Predictor Performance For Classification Applications

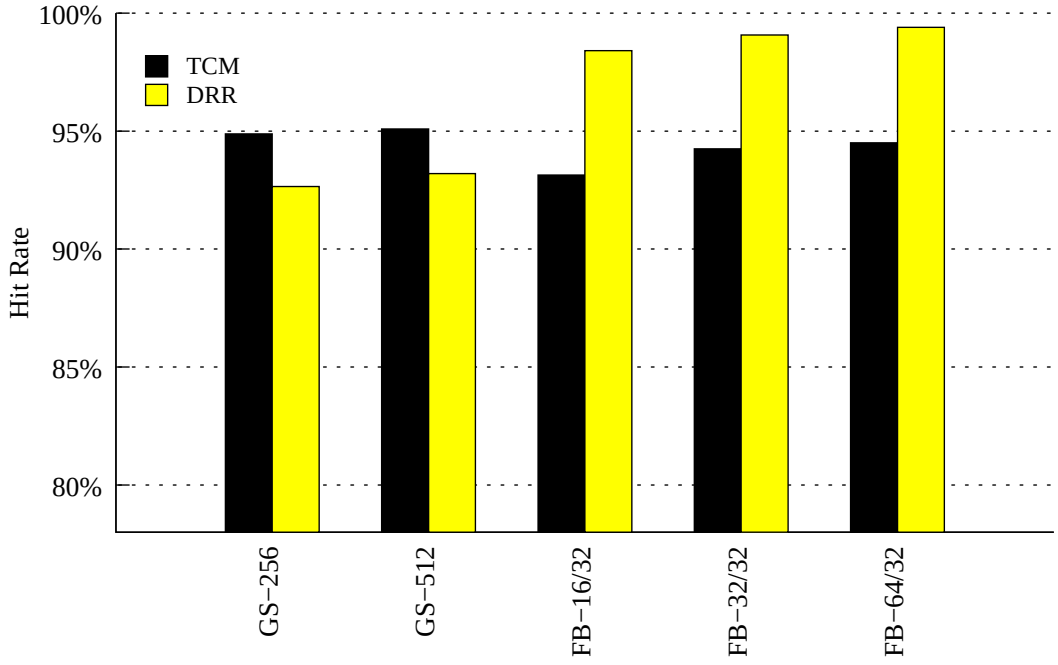


Figure 6.13: Field-Based Predictor Performance For Queueing & Metering Applications

scheme, such a mechanism would ignore the packet length and remains speculative. A more simple solution is to configure the field-based scheme in such a way that a single misprediction causes the fall-back predictor to be used, allowing some of the top level conditional code to be predicted before using the fall-back prediction. In Figure 6.13 it can be seen that a field-based scheme with a 128-entry fallback prediction provides a sim-

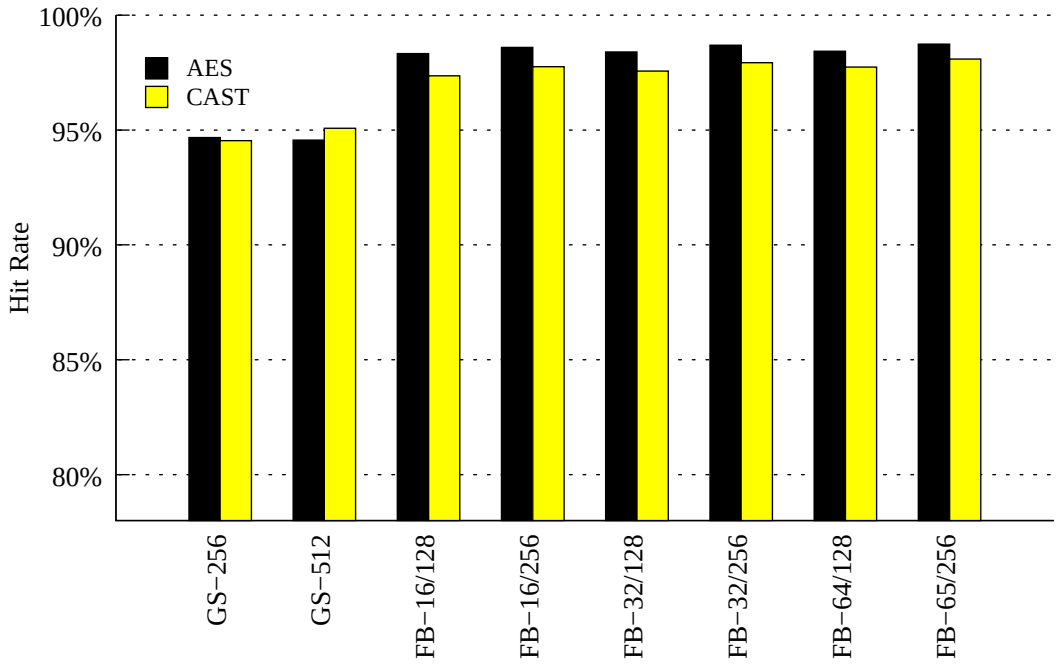


Figure 6.14: Field-Based Predictor Performance For Encryption Applications

ilar performance to a much larger *gshare* predictor, while for the DRR algorithm, average performance is increased by over 6% when comparing a 16/32 (N/M) field-based scheme and a 512-entry *gshare* predictor. Although, as with the *gshare* scheme, prediction rates for the DRR algorithm can vary when either the number of queues or the round quantum is altered.

6.6.4.2 Payload Applications

For payload applications two important factors must be noted. Firstly, since it would be inefficient to cache the entire branch history for long packets, the prediction rate after the string history has been exhausted would be largely determined by the architecture employed in the fall-back prediction. However, as was examined in section 6.4.1, two level schemes are well suited to prediction on long packets, with most payload applications (except DPI) employing a loop structure which can be easily predicted for the majority of iterations. Secondly, the percentage of small packets within the trace will heavily affect the performance of the field-based scheme. In the case of the simulations presented in this section, the TXS trace utilised has a high proportion of small packets, allowing the field-based scheme to be fully utilised. In Figure 6.14 the prediction rates for the two encryption

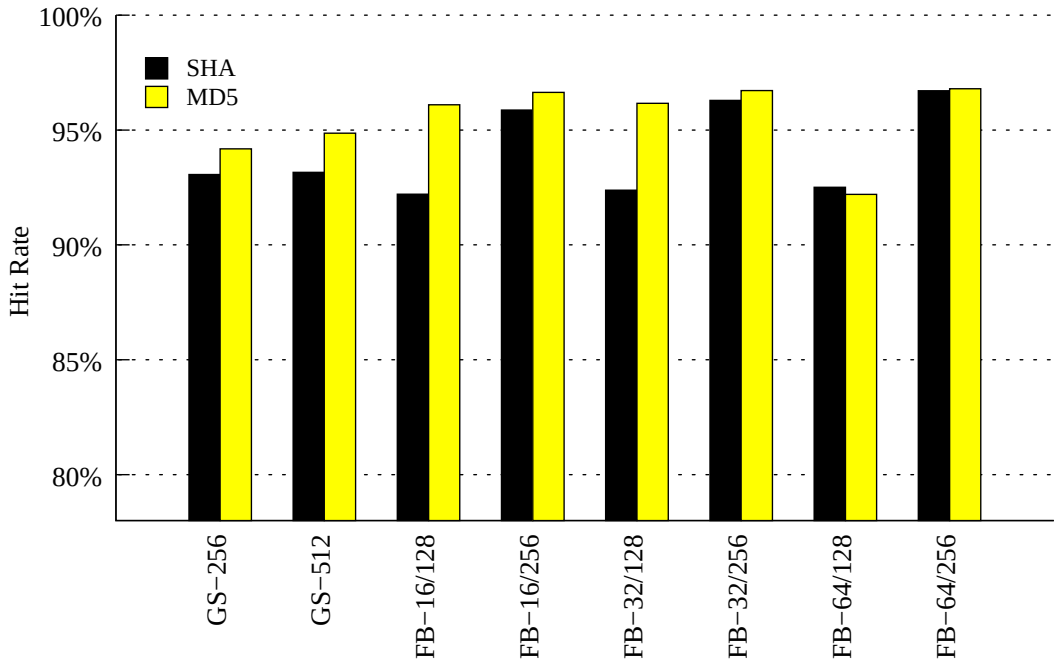


Figure 6.15: Field-Based Predictor Performance For Authentication Applications

algorithms are shown. For the field-based scheme, two configurations for each CAM size are shown, with either a 128-bit or 256-bit string table employed. Although a doubling of the string table width would be expected to significantly increase the prediction rate, it is found that in all cases the additional string history provides only small increases in performance. The high proportion of packets less than 48 bytes (121 conditional branches) means that the extra history is not fully utilised. Future sections of this chapter examine performance for other traces and highlight the expected relationship. Examining the performance of field-based schemes it can be seen that a 16 CAM entry by 128-bit field scheme outperforms a 512-entry *gshare* predictor by almost 4% (94.56%, 98.33%), while for the CAST algorithm, performance is increased from 95.08% to 97.36%. An examination of the prediction histories of these algorithms highlight a potential drawback in an evenly weighted flow-based scheme. As was discussed previously, a small number of packet lengths comprise a high proportion of all IP traffic. With payload applications allocating CAM space based on the packet length it is clear that certain packet lengths will eventually be replaced in the string history despite a heavily utilisation. A technical improvement to such a scheme might be for the programmer to allocate space within the string table for certain search keys likely to occur often.

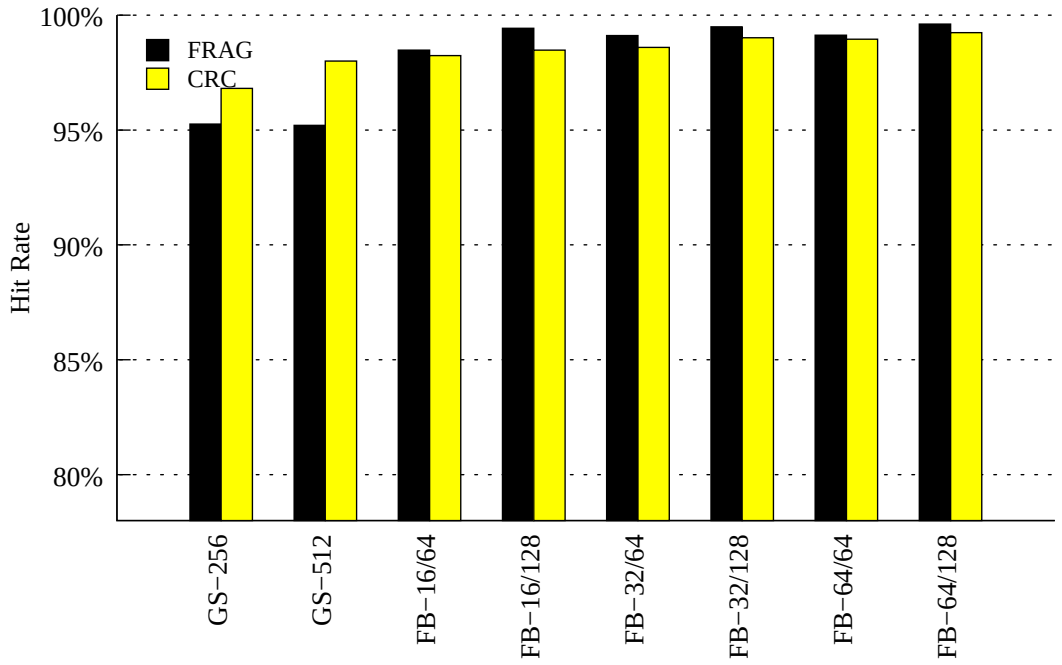


Figure 6.16: Field-Based Predictor Performance For Miscellaneous Applications

Examining the performance of authentication algorithms it can be seen how prediction rates could be improved by caching only a single packet length within the CAM structure. With only a 128 PHT entry fallback predictor used in the field-based scheme, the prediction rate for a 128-bit string table is worse than the two *gshare* architectures used for comparison due to the fact that for the SHA algorithm a minimum sized 40-byte packet requires 290 conditional operations. The MD5 algorithm is better suited to dynamic prediction than the SHA algorithm and so does improve despite the string table width being significantly less than the minimum number of conditional branches per packet (944). For the SHA algorithm, increasing the string table width to 256-bits does provide a significant performance gain since it provides almost enough branch history to cover small packets. Comparing the performance of a 16/128 field-based scheme to a larger 16/256 predictor, performance is found to increase from 92.21% to 95.87%.

The prediction performance for the two remaining payload applications, FRAG and CRC, is shown in Figure 6.16. The simplicity of the CRC algorithms ensures that a high prediction rate can be achieved with various prediction architectures. Typically implemented as a table lookup, only a single branch instruction is needed for the *main* processing of the CRC loop, while additional conditional branches are required to add the

computed CRC value to the end of the packet before updating the packet header. Since the field-based scheme retains only the initial *M* branches, the more dynamic branch operations are predicted by the fall-back predictor. For various configurations of the number of *N* (CAM entries), a 64-bit and 128-bit string table is examined. For the CRC algorithm, the high prediction rates which can be obtained with a small *gshare* predictor limit the amount of performance gain which can be achieved via a field-based scheme. On the other hand, the prediction rate for the FRAG algorithm does improve significantly when a field-based scheme is used, with a 512-entry *gshare* predictor providing a hit rate of 95.20% while the field-based predictor (16/64) obtains a hit rate of 98.48%, with a 16-entry by 128-bit configuration providing almost perfect prediction (99.43%).

Finally, the prediction rates for the combinational applications is shown in Figure 6.17. Again, the field-based scheme provides a higher prediction than any of the *gshare* configurations. Comparing a 16/256/128 field-based predictor to a 2K *gshare* predictor (not shown), we find that the performance for the four applications is 92.64%, 92.78%, 93.03% and 97.23% respectively, while the field-based scheme provides prediction rates of 96.92%, 96.92%, 94.85% and 98.84%. In terms of area requirement, a 2K *gshare* predictor requires approximately 24.5K transistors while the field-based scheme occupies 29.1K transistors.

6.6.4.3 Performance Evaluation For Other Traces

The performance of a field-based scheme follows other 'flow' based algorithms implemented on a network device. As with classification schemes or per-flow metering schemes, the average number of active flows will determine the performance. While the previous section examined performance using an OC-3 trace (TXS), this section presents a performance evaluation for a field-based scheme when routing faster OC-12 and OC-48 based connections. Intuitively it is possible to deduce that higher speed connections such as OC-12 and OC-48 links will require an increase in the number of branch histories cached within the string table, as opposed to the the string width. To evaluate this, 250,000 packets from the OC-12 AMP trace and 1,000,000 packets from the OC-48 trace were

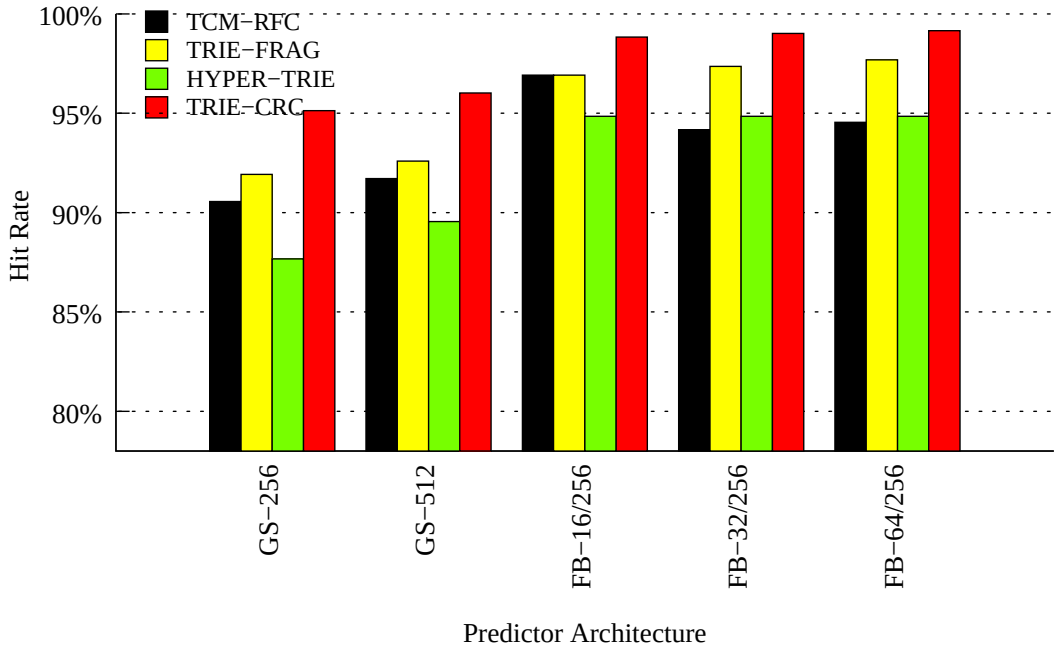


Figure 6.17: Field-Based Predictor Performance For Combinational Applications

processed by the same applications as above (Table 6.9 and Table 6.10).

For the OC-12 trace, the branch history is fixed at 128-bit for all applications. With the AMP trace containing a higher proportion of large packets, the fall-back scheme is expected to be more important for payload applications. However, as was previously examined in this chapter, the *gshare* predictor is well suited to payload applications, especially when the average packet size is high. Two configurations are examined with performance compared to a 2K *gshare* predictor. For a 32-entry scheme, prediction performance is found to be increased for 11 of the 13 applications, with all applications achieving a hit rate in excess of 95% and an average hit rate across all applications of 97.35%. Increasing the number of CAM entries to 64 allows an average prediction rate of 97.74% to be achieved. On applications such as FRAG, CAST, TRIE, HYPER, DRR and RFC, the performance gain for a 64/128/128 scheme is over 2% for each application, with increases greater than 1% seen in the AES, CRC, HASH and STAT algorithms. As was seen previously, the SHA algorithm is highly sensitive to the table size employed in the fall-back prediction. With even a 40-byte packet requiring a large amount of branch history to be cached, the majority of the prediction defaults back to 128-entry *gshare* prediction.

Table 6.9: Prediction Rate Field-Based Scheme Vs. 2K Gshare (OC-12 AMP Trace)

Application	**GS-2048**	**FB-32/128**		**FB-64/128**	
		H.R %	$\delta\%$	H.R %	$\delta\%$
AES	95.77	97.06	1.29	97.09	1.32
SHA	94.81	94.05	-0.76	94.14	-0.67
MD5	96.48	98.83	2.35	98.86	2.38
CAST	93.48	95.56	2.08	95.98	2.5
CRC	98.23	99.23	1.00	99.71	1.48
FRAG	96.56	98.88	2.32	99.24	2.68
TRIE	96.20	99.07	2.87	99.49	3.29
HASH	97.59	99.29	1.7	99.59	2
DRR	95.76	97.17	1.41	97.87	2.11
STAT	95.17	95.86	0.69	96.42	1.25
HYPER	92.84	97.44	4.6	98.25	5.41
RFC	96.77	99.35	2.58	99.53	2.76
TCM	95.86	93.82	-2.04	94.5	-1.36

Similar to the OC-12-based evaluations, 32 and 64 entry CAM configurations are compared against a 2K-entry *gshare* predictor. In both cases the string table width is fixed as 128-bits per entry. With a similar length distribution to the AMP trace, performance across the payload application is similar, with performance improvements seen in AES, CAST, CRC and FRAG algorithms. For the header applications the performance increase is smaller due to the fact that an OC-48 router would multiplex a higher number of connections during normal operation.

6.7 Conclusions

In this chapter an examination of branch prediction for NP systems was presented. For a small 5-stage pipeline the branch penalty results in a significant loss of processing cycles and with NP applications remaining in place for long periods of time, the loss in performance due to wasted microprocessor cycles must be mitigated to optimise PE performance and utilisation.

Existing branch prediction schemes fail to efficiently account for the repetitive nature

Table 6.10: Prediction Rate For OC-48 PSC Trace

Application	GS-2048	FB-32/128		FB-64/128	
		H.R %	δ%	H.R %	δ%
AES	95.77	97.34	1.57	97.4	1.63
SHA	94.68	95.17	0.49	95.5	0.82
MD5	95.86	96.66	0.8	96.76	0.9
CAST	93.78	94.49	0.71	94.5	0.72
CRC	99.14	99.68	0.54	99.68	0.54
FRAG	96.34	98.4	2.06	98.89	2.55
TRIE	95.72	98.51	2.79	98.98	3.26
HASH	97.38	99.77	2.39	99.88	2.5
DRR	96.77	98.63	1.86	99.05	2.28
STAT	95.73	95.32	-0.41	96.78	1.05
HYPER	93.2	97.37	4.17	98.99	5.79
RFC	96.62	98.19	1.57	98.34	1.72
TCM	95.96	94.81	-1.15	94.62	-1.34

of NP applications. Various NP applications only achieve high prediction rates when large PHT are employed, but since NP applications are generally small applications the vast majority of these table entries remain idle for long periods of time. In addition to this under-utilisation, existing schemes such as *gshare* fail to take into account NP specific application level information when deciding if a branch will be taken or not.

To improve prediction rates, a field-based prediction scheme is proposed, which incorporates per-packet information within the branch predictor. Incorporating packet level information at the time of prediction, the field-based scheme attempts to utilise flow-level history as a means of guiding future prediction decisions. Exploiting common network behaviour and topologies, the field-based scheme improves the branch prediction performance of many NP applications, with the exception of only inter-arrival-based applications such as metering applications (e.g. TCM). Being well suited to highly conditional code, the field-based scheme significantly improves performance on common header based applications such as forwarding or packet classification. Furthermore, while achievable prediction rates are limited for payload based applications, modern NP platforms have increasingly moved such data intensive algorithms to hardware, leaving the

PE to perform the conditional aspect of the applications. The field-based predictor can also be scaled in order to meet the requirements of an NP operating at various line rates and it is particularly well suited to highly dynamic network loads.

CHAPTER 7

Conclusions & Future Work

This section summarises the research goals of the author, presents an examination and discussion of the results achieved, as well as briefly examining possible future directions of research within the scope of PE design for network processors.

7.1 Motivation for Proposed Research – A Summary

Examining the modern Internet it is clear that a number of trends are apparent which pose significant challenges to network designers and researchers. Firstly, the number of devices connected to the global Internet has increased exponentially. As well as connecting an increasing number of users via traditional PC based systems, the difference between existing mobile communication systems and IP based packet switched networks is narrowing, with 'Anywhere, Anytime' networks becoming more common. Network technologies such as 3G and 4G can be viewed as packet switched networks optimised for mobile communications. Devices such as Apple's iPhone and Amazon's Kindle E-Book reader all provide a means of connecting to either mobile networks (GSM, EDGE, CDMA, etc.) or IEEE802.11 'Wi-Fi' wireless networks. Along with the trends towards larger, more diverse and higher bandwidth networks is the evolution in the functions provided by IP networks. Tasks such as packet forwarding remain fundamental but form only a base on which quality and security services are also provided. Provision of services such

as VOIP, IPTV or video conferencing require networks to be aware of both packet flows and packet content, detecting high priority content within a heavily multiplexed network flow. Even providing more basic services, such as a fair usage policy, requires ISPs to detect, classify and meter those users which are using more than their fair share of network resources.

For network researchers, the trends outlined above can be summarised as follows. Future networks must connect more users, at higher speeds, while also allowing network providers to add and remove services as user demands change. For a programmable NP system, the challenge is how the reprogrammable aspect of router design can be retained while also increasing performance. In Chapter 2 it was outlined how techniques to increase NP performance have typically involved CMOS technological improvements, increased parallelism or the offloading of specific functions to dedicated hardware. In the first case it is clear that technology increments cannot be relied upon to provide future performance increases. On the other hand, hardware offloading provides a method of increasing performance but at the cost of less flexibility. Numerous hardware architectures have been proposed for NP tasks such as packet forwarding, five tuple classification or deep packet inspection but all three of these topics have been actively researched (and improved), highlighting the challenges with hardware specific solutions. In addition to technological changes and hardware offloading, another method by which NP performance can be increased is to parallelise the NP architecture by implementing a larger number of PEs. Parallelisation does however provide a number of significant challenges such as memory and I/O bandwidth latency, load balancing and software programmability.

Examining each mechanism it is clear that future NP designs will have to strike a balance between these techniques and more fundamental micro-architectural considerations such as pipeline depth. With these considerations in mind, the goal of this research was to investigate methods of improving NP performance by examining the behaviour and performance of branch operations within an NP system. Whereas previous published work has examined topics such as caching, the goal of the author's research was to determine if deeper processor pipelines could be implemented while avoiding the traditional limi-

tation associated with these deeper pipelines, namely the branch penalty incurred during conditional operations.

7.2 Summary of Thesis Contributions

The context for the work presented in this thesis was briefly summarised in the previous section, with Chapter 1 providing a more broad discussion of the work described in this thesis. The author's goals and objectives were also outlined in Chapter 1. Chapter 2 presented a high level technical background on the topic of network processors before presenting an overview of the concept of pipeline processors, pipeline hazards and the various prediction techniques used to overcome the control hazard. Chapter 3 presented a discussion of performance evaluation methods and metrics for NP architectures. It was argued that the lack of a coherent NP simulation framework seriously limited the degree of research regarding architectural performance which can be undertaken, motivating the need for a new NP simulator.

7.2.1 The SimNP NP Simulator

In Chapter 4 a framework for modelling NP systems was presented. A significant limitation within NP research, the lack of a flexible and open source system simulator means that realistic comparisons between existing work is almost impossible. For example, per packet processing rates of external hardware accelerators commonly fail to take into account limitations such as contention or configuration time, making performance figures idealistic and difficult to evaluate. Existing NP system simulators are found to be either too specific to one architecture or to require extensive configuration and development in order to build the required architecture. High level modelling via Petri-Nets or a queue model allows certain performance figures to be extracted but require either a large amount of input knowledge (request/service rate), or can only be implemented by simplifying a significant part of the system to be modelled. The proposed simulator is designed using a single language and utilises a flat-memory model, allowing memory mapped devices

to be rapidly added/removed from the simulation framework. It can be programmed in a single high level language such as *C* and can be easily re-ordered to create various configurations. A primary goal of the SimNP simulator was to provide an open source NP simulator which could be developed with ease in order to better model NP platforms. With this in mind a number of possible additions are briefly outlined which the author believes would improve the usability and precision of the SimNP simulator. Improvements to the programming framework and the addition of debugging facilities would allow faster application development. At a hardware level, further work could improve the accuracy by implementing more detailed memory models, e.g. DDR DRAM, QDR SRAM, etc., allowing aspects such as non-deterministic latency and paging to be investigated across memory units. Also, a more complete interface model would allow entire line-cards to be simulated.

7.2.2 Workload Analysis and Branch Behaviour of NP Applications

Using the SimNP simulator, a workload analysis of NP applications was presented in Chapter 5. Topics such as memory distribution, parallelism, processing time and branch behaviour were examined. For flexible software stack pointer architectures such as the ARM architecture, local RAM was found to be extremely important to NP performance. Comprising a majority of all memory operations, it is clear that access latency between the PE and any local RAM must be minimised, pointing to the fact that it may not be possible to share local RAM between multiple PEs and it almost certainly would not be possible to move the local RAM space within an NP design to external memory. With respect to parallelism, performance increases above 16 PEs are found to be smaller when compared to a change from 8 to 16 PEs. High device contention limits the utilisation of each PE, a factor which would be made worse by techniques such as multi-threading. The final aspect of the workload analysis was to examine the behaviour of conditional branch operations within NP applications. While the ratio of taken to not-taken branches varies greatly across NP applications, a number of important NP specific points can be obtained. Firstly, despite the small application size, the number of cycles lost through conditional

branches in an NP system is large since NP applications process high volumes of data, almost continually, for long periods of time. Secondly, this performance loss is due to only a small amount of unique branch instructions (unique in either a temporal or spatial location).

7.2.3 Branch Prediction for Network Processors

Finally, in Chapter 6 a detailed analysis of branch prediction for NP systems was presented. Although more static than general purpose systems in the sense that an NP application will remain deployed for a long period of time, the randomness within NP traffic is found to limit the performance of many existing prediction schemes. Variance within the underlying traffic makes static branch prediction difficult to implement, despite the small application kernel. Dynamic prediction schemes are found to provide prediction rates in excess of 90%, but only when the number of pattern history entries is large. Viewing the problem from a different perspective, it is clear that it should be possible to achieve very high prediction rates within a programmable NP system. While certain network characteristics such as packet length or inter-arrival time are random between one packet and another, the data unit processed by a PE does not change. Furthermore, there are certain aspects within network traffic which, although difficult to predict from one packet to another, can be seen across either a section of the network load or indeed, the entire network history. It is with this in mind that the Field-Based branch predictor outlined in Chapter 5 was presented. Whereas existing dynamic predictors utilise only run-time history in future branch predictions, the Field-Based scheme uses both run-time history and packet level information when deciding if a branch will likely be taken or not. Capable of being configured at run-time, the Field-Based predictor was scalable in terms of performance and area utilisation, and was found to compare favourably to existing methods in terms of prediction hit rates, area and latency.

7.3 Future Work

Within the scope of NP design there are a number of principal areas that will require future research. Firstly, with parallelism and multi-threading increasingly common within NP design, current programming languages represent a significant limitation. Commercial NPs typically provide programming frameworks adapted from a standard language such as C. While such modifications allow parallelism to be exploited, a pseudo-c programming language slows software development by limiting the scope of the existing code base which can be used, while making the design and implementation of re-usable, architecture agnostic software difficult. Future standardisations of both the C and C++ languages should include mechanisms for programming paradigms such as parallelism, but with NP architectures highly sensitive to latency it may be difficult to fully utilise constructs optimised for general purpose systems. With reference to the work presented in this thesis, the evaluation of new programming frameworks can easily be achieved via SimNP. The PEs utilise the ARM instruction set, allowing mature compilers for various languages to be exploited (*C*, *C++*). Furthermore, while a workload analysis provides a good method of classifying, analysing and quantifying performance of various NP applications, the lack of a good programming framework limits the amount of analysis which can be achieved under more realistic conditions, i.e. analysis of a fully programmed line card.

The work presented in Chapter 6 represents one possible method of improving PE performance without sacrificing the flexibility of an NP system. Other areas of research outside the author's core topics might include; improvements to how a cache hierarchy is employed in systems with low spatial data locality and a NP specific superscalar architecture which follows the same design flow as the field-based prediction scheme proposed in this work, i.e. by tailoring an existing design technique to an NP system.

BIBLIOGRAPHY

[1] K. G. Coffman and A. M. Odlyzko, "Internet Growth: Is there a Moores Law for Data Traffic?', Handbook of Massive Data Sets," in *Handbook of Massive Data Sets*. Kluwer, 2001, pp. 47–93.

[2] A. Odlyzko, "Internet Growth: Myth and Reality, Use and Abuse," *Journal of Computer Resource Management*, vol. 102, pp. 23–27, 2001.

[3] L. Roberts, "Beyond Moore's Law: Internet Growth Trends," *Computer, IEEE*, vol. 33, no. 1, pp. 117–119, Jan 2000.

[4] Minnesota Internet Traffic Studies (MINTS). [Online]. Available: http://www.dtc.umn.edu/mints/home.php

[5] Cisco Visual Networking Index - Forecast and Methodology, 2007-2012. Cisco Inc. [Online]. Available: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705%/ns827/white_paper_c11-481360.pdf

[6] The Economist, "Connecting The Dragon." [Online]. Available: http://www.economist.com/daily/chartgallery/displaystory.cfm?story_id=1%1831676

[7] S. Cherry, "Nothing But Net," *Spectrum, IEEE*, vol. 44, no. 1, pp. 22–26, Jan. 2007.

[8] E. Svoboda, "One-Click Content, No Guarentees," *IEEE Spectrum*, vol. 43, pp. 64–65, 2006.

[9] F. Douglis, "Staring at Clouds," *Internet Computing, IEEE*, vol. 13, no. 3, pp. 4–6, May-June 2009.

[10] D. Milojicic, "Cloud Computing: Interview with Russ Daniels and Franco Travostino," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 7–9, Sept.-Oct. 2008.

[11] C. Petrie and C. Bussler, "The Myth of Open Web Services: The Rise of the Service Parks," *Internet Computing, IEEE*, vol. 12, no. 3, pp. 96–9, May-June 2008.

[12] N. Leavitt, "Is Cloud Computing Really Ready for Prime Time?" *Computer, IEEE*, vol. 42, no. 1, pp. 15–20, Jan. 2009.

[13] YouTube Inc. [Online]. Available: http://www.youtube.com

[14] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "YouTube Traffic Characterization: A View From The Edge," in *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2007, pp. 15–28.

[15] S. Kumar, J. Chhugani, C. Kim, D. Kim, A. Nguyen, P. Dubey, C. Bienia, and Y. Kim, "Second Life and the New Generation of Virtual Worlds," *Computer, IEEE*, vol. 41, no. 9, pp. 46–53, Sept. 2008.

[16] Q. Zhu, T. Wang, and Y. Jia, "Second Life: A New Platform for Education," in *Information Technologies and Applications in Education, 2007. ISITAE '07. First IEEE International Symposium on*, Nov. 2007, pp. 201–204.

[17] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross, "A Measurement Study of a Large-Scale P2P IPTV System," *Multimedia, IEEE Transactions on*, vol. 9, no. 8, pp. 1672–1687, Dec. 2007.

[18] S. Sen and J. Wang, "Analyzing Peer-To-Peer Traffic Across Large Networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 2, pp. 219–232, 2004.

[19] Spotify ltd. [Online]. Available: http://www.spotify.com

[20] H. Zimmermann, "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980.

[21] S. Borkar, N. P. Jouppi, and P. Stenstrom, "Microprocessors in the Era of Terascale Integration," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 237–242.

[22] MIPS 32 and 64-bit Cores. MIPS Inc. [Online]. Available: http://www.mips.com/products/processors/23-64-bit-cores/

[23] S. Furber, *ARM System-on-Chip Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[24] ARC 710D Core. ARC Ltd. [Online]. Available: http://www.arc.com/configurablecores/arc700/710.html

[25] Intel Xscale Technology. Intel Inc. [Online]. Available: http://www.intel.com/design/intelxscale

[26] V. Cerf and R. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications*, vol. 22, pp. 637–648, 1974.

[27] K. Thompson, G. Miller, and R. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics," *Network, IEEE*, vol. 11, no. 6, pp. 10–23, Nov/Dec 1997.

[28] M. Fomenkov, K. Keys, D. Moore, and K. Claffy, "Longitudinal Study of Internet Traffic in 1998-2003," in *WISICT '04: Proceedings of the winter international symposium on Information and communication technologies*. Trinity College Dublin, 2004, pp. 1–6.

[29] CAIDA Network Project - , "CAIDA Long Term Workload Trend Analysis," http://www.caida.org/publications/papers/2000/AIX0005/AIX0005.pdf (accessed on 20090316).

[30] S. V. Kartalopoulos, *Communications Networks for the Next Mellennium*. Wiley-IEEE, 1999.

[31] D. Cavendish, "Evolution of Optical Transport Technologies: From SONET/SDH to WDM ," *Communications Magazine, IEEE*, vol. 38, no. 6, pp. 164–172, Jun 2000.

[32] K. Raza and M. Turner, *Large-Scale IP Network Solutions (CCIE Professional Development)*. Indianapolis, IN: Cisco Press, 2002, ch. 3.

[33] Light Reading, "Sprint Trials 40G." [Online]. Available: http://www.lightreading.com/document.asp?doc_id=168439

[34] A. Adiletta, M. Rosenbluth, and D. Bernstein, "The Next Generation of Intel IXP Network Processors," *Intel Technology Journal*, vol. 6, no. 3, 2002.

[35] R. Cam, M. Tuck, R. Lerer, K. Gass, and W. Nation, "System Packet Interface Level 4 (SPI-4) Phase 2," Optical Internetworking Forum, CA, Tech. Rep. OIF-SPI-4-02.1, Oct 2003.

[36] M. R. Hussain, "Cavium OCTEON Multi-Core Processor," *Keynote presented at the 2006 ACM/IEEE Symposium on Architecture for Networking and Communication Systems, ANCS 2006.*, 2006.

[37] P. Deutsch and J. Gailly, "ZLIB Compressed Data Format Specifications version 3.3," in *RFC 1950*, United States, 1998. [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt

[38] P. Lekkas, "EZchip TOPcore," in *Network Processors: Architectures, Protocols and Platforms*. New York, NY, USA: McGraw-Hill, Inc., 2003, ch. 6, pp. 148–152.

[39] "Xelerator X11 Network Processor - Product Brief," Xelerated, Sweden, Tech. Rep., Jan 2008. [Online]. Available: http://www.xelerated.com/uploads/files/5.PDF

[40] "5NP4G Network Processor - Product Brief," HIFN Inc., USA, Tech. Rep., Jan 2008. [Online]. Available: http://www.hifn.com/uploadedFiles/Library/Product_Briefs/5NP4G_pb_v1.pd%f

[41] "NFP-3200 Network Flow Processor," Netronome Systems Inc., USA, Tech. Rep., Mar 2009. [Online]. Available: http://www.netronome.com/files/file/Netronome%20NFP%20Product%20Brief%2%0(3-09).pdf

[42] "The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor," Cisco Corporation., USA, Tech. Rep., Aug 2007. [Online]. Available: http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_over%view_c22-448936.pdf

[43] EZ-Chip Technologies. (2009). [Online]. Available: http://www.ezchip.com

[44] W. Eatherton, "The Push of Network Processing to the Top of the Pyramid," *Keynote presented at the 2005 ACM/IEEE Symposium on Architecture for Networking and Communication Systems, ANCS 2005.*, 2005.

[45] N. Weng and T. Wolf, "Analytic Modeling of Network Processors for Parallel Workload Mapping," *ACM Transactions on Embedded Computing Systems*, vol. 8, no. 3, pp. 1–29, 2009.

[46] F. Karim, A. Nguyen, S. Dey, and R. Rao, "On-Chip Communication Architecture for OC-768 Network Processors," in *DAC '01: Proceedings of the 38th conference on Design automation*. New York, NY, USA: ACM, 2001, pp. 678–683.

[47] D. Flynn, "AMBA: Enabling Reusable On-Chip Designs," *Micro, IEEE*, vol. 17, no. 4, pp. 20–27, Jul/Aug 1997.

[48] T. Ainsworth and T. Pinkston, "Characterizing the Cell EIB On-Chip Network," *Micro, IEEE*, vol. 27, no. 5, pp. 6–14, Sept.-Oct. 2007.

[49] A. Chame, "PCI Bus in High Speed I/O Systems Applications," in *Aerospace Conference, 1998. Proceedings., IEEE*, vol. 4, Mar 1998, pp. 505–514 vol.4.

[50] M. Peyravian and J. Calvignac, "Fundamental Architectural Considerations for Network Processors," *Computer Networks.*, vol. 41, no. 5, pp. 587–600, 2003.

[51] J. Mudigonda, H. Vin, and R. Yavatkar, "Managing Memory Access Latency in Packet Processing," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2005, pp. 396–397.

[52] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*, D. Penrose, Ed. Morgan Kaufmann, 2003.

[53] A. Kennedy, X. Wang, and B. Liu, "Energy Efficient Packet Classification Hardware Accelerator," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–8.

[54] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet inspection," in *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2006, pp. 339–350.

[55] X. Meng, Z. Xu, B. Zhang, G. Huston, S. Lu, and L. Zhang, "IPv4 Address Allocation and the BGP Routing Table Evolution," *SIGCOMM Computer Communications Review*, vol. 35, no. 1, pp. 71–80, 2005.

[56] BGP Routing Table Analysis. [Online]. Available: http://bgp.potaroo.net

[57] K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," in *RFC 1631*, United States, 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1631.txt

[58] P. Gupta, "Algorithms for Routing Lookups and Packet Classifications," Ph.D. dissertation, Stanford Univ., Stanford, December 2000. [Online]. Available: http://klamath.stanford.edu/~pankaj/phd.html

[59] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2003, pp. 213–224.

[60] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM, 1999, pp. 147–160.

[61] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM, 1999, pp. 135–146.

[62] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an Alternative to CAMs?" in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, vol. 1, March-3 April 2003, pp. 53–63 vol.1.

[63] Skype Inc. [Online]. Available: http://www.skype.com

[64] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli, "Revealing Skype Traffic: When Randomness Plays With You," in *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2007, pp. 37–48.

[65] R. Oppliger, "Security at the Internet layer," *Computer, IEEE*, vol. 31, no. 9, pp. 43–47, Sep 1998.

[66] S. A. Thomas, *SSL and TLS Essentials: Securing the Web*. New York, NY, USA: John Wiley & Sons, Inc., 2000.

[67] J. Daemen and V. Rijmen, *The Design of Rijndael*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002.

[68] National Institute of Standards and Technology (NIST), U.S. Dept. of Commerce, "FIPS 180-2: Secure Hash Standard (SHS)," Tech. Rep. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenot%ice.pdf

[69] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao, "A Fast String-Matching Algorithm for Network Processor-Based Intrusion Detection System," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 3, pp. 614–633, 2004.

[70] M. Becchi and P. Crowley, "Efficient Regular Expression Evaluation: Theory to Practice," in *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. New York, NY, USA: ACM, 2008, pp. 50–59.

[71] Y. H. Cho and W. H. Mangione-Smith, "Deep Network Packet Filter Design for Reconfigurable Devices," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 2, pp. 1–26, 2008.

[72] J. Guo, F. Chen, L. Bhuyan, and R. Kumar, "A Cluster-based Active Router Architecture Supporting Video/Audio Stream Transcoding Service," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, April 2003, pp. 8 pp.–.

[73] P. Crowley, M. E. Fluczynski, J.-L. Baer, and B. N. Bershad, "Characterizing Processor Architectures for Programmable Network Interfaces," in *ICS '00: Proceedings of the 14th international conference on Supercomputing*. New York, NY, USA: ACM, 2000, pp. 54–65.

[74] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer, "Exploring Trade-offs in Performance and Programmability of Processing Element Topologies for Network Processors," in *2nd Workshop on Network Processors (NP2) at the 9th International Symposium on High Performance Computer Architecture (HPCA9), Anaheim CA*, February 2003, pp. 75–87. [Online]. Available: http://www.gigascale.org/pubs/354.html

[75] H. Liu, "A Trace Driven Study of Packet Level Parallelism," in *Communications, 2002. ICC 2002. IEEE International Conference on*, vol. 4, 2002, pp. 2191–2195 vol.4.

[76] L. Shi, Y. Zhang, J. Yu, B. Xu, B. Liu, and J. Li, "On the Extreme Parallelism Inside Next-Generation Network Processors," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, May 2007, pp. 1379–1387.

[77] S. Melvin and Y. Patt, "Handling of Packet Dependencies: A Critical Issue for Highly Parallel Network Processors," in *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2002, pp. 202–209.

[78] Y. Qi, B. Xu, F. He, X. Zhou, J. Yu, and J. Li, "Towards Optimized Packet Classification Algorithms for Multi-Core Network Processors," in *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2007, p. 2.

[79] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, "Towards High-Performance Flow-Level Packet Processing on Multi-Core Network Processors," in *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2007, pp. 17–26.

[80] S. Govind, R. Govindarajan, and J. Kuri, "Packet Reordering in Network Processors," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007, pp. 1–10.

[81] D. Bermingham, X. Wang, and L. Bin, "Analysis of FPGA-based AES round architectures," in *Irish Signals and Systems Conference, 2005. IET Conference*, Dublin, Ireland, 2005, pp. 324 – 329.

[82] X. Zhang and K. Parhi, "High-Speed VLSI architectures for the AES algorithm," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 9, pp. 957–967, Sept. 2004.

[83] LEON2 SPARC Compatible Microprocessor. Gaisler Inc. [Online]. Available: http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=12%&Itemid=52

[84] C.-P. Su, T.-F. Lin, C.-T. Huang, and C.-W. Wu, "A High-Throughput Low-Cost AES Processor," *Communications Magazine, IEEE*, vol. 41, no. 12, pp. 86–91, Dec. 2003.

[85] M. Bohr, "Intel's 90 nm Technology : Moore's Law and More," in *Intel Developer Forum*, September 2002.

[86] T. Skotnicki, J. Hutchby, T.-J. King, H.-S. Wong, and F. Boeuf, "The End of CMOS Scaling: Toward the Introduction of New Materials and Structural Changes to Improve MOSFET Performance," *Circuits and Devices Magazine, IEEE*, vol. 21, no. 1, pp. 16–26, Jan.-Feb. 2005.

[87] W. M. Johnson, "Super-Scalar Processor Design," Stanford, CA, USA, Tech. Rep., 1989. [Online]. Available: ftp://reports.stanford.edu/pub/cstr/reports/csl/tr/89/383/CSL-TR-89-383%.pdf

[88] J. Shen, M. Lipasti, and J. Shen, *Modern Processor Design: Fundamentals of Superscalar Processors*. USA: McGraw-Hill, 2004.

[89] D. W. Wall, "Limits of Instruction-Level Parallelism," in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1991, pp. 176–188.

[90] G. Memik and W. H. Mangione-Smith, "Evaluating Network Processors using NetBench," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 453–471, 2006.

[91] K. L. Byeong and K. J. Lizy, "NpBench: A Benchmark Suite for Control plane and Data plane Applications for Network Processors," in *ICCD '03: Proceedings of the 21st International Conference on Computer Design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 226.

[92] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, vol. Q1, 2001.

[93] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 5, pp. 525–539, Sep 1992.

[94] H. Che, C. Kumar, and B. Menasinahal, "Fundamental Network Processor Performance Bounds," in *Network Computing and Applications, Fourth IEEE International Symposium on*, July 2005, pp. 179–185.

[95] S. Ramakrishna and H. Jamadagni, "Analytical Bounds on the Threads in IXP1200 Network Processor," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, Sept. 2003, pp. 426–429.

[96] C. Ostler, K. S. Chatha, V. Ramamurthi, and K. Srinivasan, "ILP and Heuristic Techniques for System-level Design on Network Processor Architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 4, p. 48, 2007.

[97] P. Lekkas, "Network Processors: Justification," in *Network Processors: Architectures, Protocols and Platforms*. New York, NY, USA: McGraw-Hill, Inc., 2003, ch. 2, pp. 36–37.

[98] D. Comer, *Network Systems Design using Network Processors*. NJ, USA: Pearson Education Inc., 2005, ch. 14, p. 210.

[99] Z. Liu, J. Yu, X. Wang, B. Liu, and L. Bhuyan, "Revisiting the Cache Effect on Multicore Multithreaded Network Processors," in *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, Sept. 2008, pp. 317–324.

[100] T. Wolf and M. Franklin, "CommBench - A Telecommunications Benchmark for Network Processors," in *ISPASS '00: Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 154–162.

[101] J. Mudigonda, H. M. Vin, and R. Yavatkar, "Overcoming the Memory Wall in Packet Processing: Hammers or Ladders?" in *ANCS '05: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2005, pp. 1–10.

[102] M. Hill and M. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33 –38, july 2008.

[103] GNU. (2009) GNU Compiler collection. [Online]. Available: http://gcc.gnu.org

[104] MSP430 16-bit Ultra-Low Power MCUs. Texas Instruments Inc. [Online]. Available: http://focus.ti.com/mcu/docs/mcuprodoverview.tsp?sectionId=95&tabId=140%&familyId=342

[105] J. E. Smith, "A Study of Branch Prediction Strategies," in *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 135–148.

[106] S. McFarling and J. Hennesey, "Reducing the Cost of Branches," in *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 396–403.

[107] R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly, "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks," *ACM SIGOPS*, vol. 25, no. Special Issue, pp. 290–302, 1991.

[108] J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," in *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1992, pp. 85–95.

[109] C. Young and M. D. Smith, "Static Correlated Branch Prediction," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 1028–1075, 1999.

[110] N. Gloy, M. Smith, and C. Young, "Performance Issues in Correlated Branch Prediction Schemes," in *Microarchitecture, 1995. Proceedings of the 28th Annual International Symposium on*, Nov-1 Dec 1995, pp. 3–14.

[111] D. W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles," in *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1991, pp. 59–70.

[112] H. Jiang and C. Dovrolis, "Why is the Internet Traffic Bursty in Short Time Scales?" *SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 241–252, 2005.

[113] T. Ball and J. R. Larus, "Branch Prediction for Free," in *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1993, pp. 300–313.

[114] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. Hwu, "SuperBlock Formation Using Static Program Analysis," in *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 247–255.

[115] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-Based Static Branch Prediction Using Machine Learning," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, pp. 188–222, 1997.

[116] B. Deitrich, B. C. Chen, and W. Hwu, "Improving Static Branch Prediction in a Compiler," in *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, Oct 1998, pp. 214–221.

[117] R. Gupta, D. A. Berson, and J. Z. Fang, "Path Profile Guided Partial Redundancy Elimination Using Speculation," in *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*. Washington, DC, USA: IEEE Computer Society, 1998, p. 230.

[118] T.-Y. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pp. 124–134, 1992.

[119] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer.*, vol. 17, no. 1, pp. 6–22, Jan 1984.

[120] T. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *IEEE Instruction-level parallel processors*, pp. 150–160, 1995.

[121] S. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," in *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1992, pp. 76–84.

[122] S. McFarling, "Combining Branch Predictors," Tech. Rep. TN-36, June 1993. [Online]. Available: ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TN-36.pdf

[123] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," in *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 295–306.

[124] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 284–291, 1997.

[125] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The Bi-Mode Branch Predictor," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium*

on Microarchitecture. Washington, DC, USA: IEEE Computer Society, 1997, pp. 4–13.

[126] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," in *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 69–77.

[127] D. Burger, T. M. Austin, and S. W. Keckler, "Recent Extensions to the SimpleScalar Tool Suite," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 4–7, 2004.

[128] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2000, pp. 83–94.

[129] J. Edler and M. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator." [Online]. Available: http://www.cs.wisc.edu/~markhill/DineroIV/

[130] J. Eckberg, A., "The single server queue with periodic arrival process and deterministic service times," *Communications, IEEE Transactions on*, vol. 27, no. 3, pp. 556 – 562, mar 1979.

[131] M. Marsan, G. Balbo, and G. Conte, "Comparative Performance Analysis of Single Bus Multiprocessor Architectures," *Computers, IEEE Transactions on*, vol. C-31, no. 12, pp. 1179–1191, Dec. 1982.

[132] B. Bodnar and A. Liu, "Modeling and Performance Analysis of Single-Bus Tightly-Coupled Multiprocessors," *Computers, IEEE Transactions on*, vol. 38, no. 3, pp. 464–470, Mar 1989.

[133] T.-F. Tsuei and M. Vernon, "A Multiprocessor Bus Design Model Validated by System Measurement," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 6, pp. 712–727, Nov 1992.

[134] I. Bucher and D. Calahan, "Models of Access Delays in Multiprocessor Memories," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 3, pp. 270–280, May 1992.

[135] T. Wolf and M. A. Franklin, "Performance Models for Network Processor Design," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 6, pp. 548–561, 2006.

[136] R. Zurawski and M. Zhou, "Petri Nets and Industrial Applications: A Tutorial," *Industrial Electronics, IEEE Transactions on*, vol. 41, no. 6, pp. 567–583, Dec 1994.

[137] S. Govind and R. Govindarajan, "Performance Modeling and Architecture Exploration of Network Processors," in *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, Sept. 2005, pp. 189–198.

[138] R. E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[139] E. Larson, S. Chatterjee, and T. Austin, "MASE: A Novel Architecture or Detailed Microarchitectural Modeling," in *International Symposium on Performance Analysis of Systems and Software, 2001, (ISPASS-2001). IEEE International Symposium on*, Nov 2001.

[140] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, "RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors ," *Computer*, vol. 35, no. 2, pp. 40–49, Feb 2002.

[141] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: A Memory System Simulator," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 100–107, 2005.

[142] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "SIMICS: A Full System Simulation Platform," *Computer, IEEE*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[143] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modelling and Compututer Simulation*, vol. 7, no. 1, pp. 78–103, 1997.

[144] R. Ramaswamy and T. Wolf, "PacketBench: A Tool for Workload Characterization of Network Processing," in *Workload Characterization, 2003. WWC-6. 2003 IEEE International Workshop on*, Oct. 2003, pp. 42–50.

[145] L. Bhuyan and H. Wang, "Execution-driven Simulation of IP Router Architectures," in *Network Computing and Applications, 2001. NCA 2001. IEEE International Symposium on*, 2001, pp. 145–155.

[146] D. Suryanarayanan, J. Marshall, and G. T. Byrd, "A Methodology and Simulator for the Study of Network Processors," in *Network Processor Design : Issues and Practices*. San Francisco, CA, USA: Morgan Kaufmannn Publishers, 2003, vol. 1, pp. 27–54.

[147] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transaction on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[148] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: a System-level Exploration Platform for Network Processors," *Design & Test of Computers, IEEE*, vol. 19, no. 6, pp. 17–26, Nov/Dec 2002.

[149] T. Wild, A. Herkersdorf, and R. Ohlendorf, "Performance Evaluation for System-on-Chip Architectures Using Trace-based Transaction Level Simulation," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, March 2006, pp. 1–6.

[150] Y. Wang and X. Zhang, "A Novel Modeling Method of Network Processor Architecture Based on SystemC," in *Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference on*, Sept. 2006, pp. 109–109.

[151] Y. Luo, J. Yang, L. Bhuyan, and L. Zhao, "NePSim: A Network Processor Simulator with a Power Evaluation Framework," *Micro, IEEE*, vol. 24, no. 5, pp. 34–44, Sept.-Oct. 2004.

[152] Q. Wang, J. Chen, W. Zhang, M. Yang, and B. Zang, "Optimizing Software Cache Performance of Packet Processing Applications," in *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM, 2007, pp. 227–236.

[153] P. Reiher., "Experiences with the Intel IXP1200," UCLA, CA, USA, Tech. Rep., 2003. [Online]. Available: www.ixaedu.com/events/EducationSummit2003/UCLA.pdf

[154] K. Lee, "OpenNP: A Generic Programming Model For Network Processors," Ph.D. dissertation, University of Lancaster., Lancaster, May 2007. [Online]. Available: http://www.kevin-lee.co.uk/viva.ppt

[155] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A Productive Software Development Approach for Network Processors," *IEEE Micro*, vol. 24, no. 5, pp. 45–54, 2004.

[156] L. Li, B. Huang, J. Dai, and L. Harrison, "Automatic Multithreading and Multiprocessing of C programs for IXP," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2005, pp. 132–141.

[157] J. Dai, B. Huang, L. Li, and L. Harrison, "Automatically Partitioning Packet Processing Applications for Pipelined Architectures," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 237–248.

[158] S. Meijer, B. Kienhuis, J. Walters, and D. Snuijf, "Automatic Partitioning and Mapping of Stream-Based Applications onto the Intel IXP Network Processor,"

in *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*. New York, NY, USA: ACM, 2007, pp. 23–30.

[159] R. Ramaswamy, N. Weng, and T. Wolf, "Analysis of Network Processing Workloads," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, March 2005, pp. 226–235.

[160] H. Xie, L. Zhao, and L. Bhuyan, "Architectural Analysis and Instruction-Set Optimization Design of Network Protocol Processors," in *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, Oct. 2003, pp. 225–230.

[161] Z. Tan, C. Lin, H. Yin, and B. Li, "Optimization and Benchmark of Cryptographic Algorithms on Network Processors," *IEEE Micro*, vol. 24, no. 5, pp. 55–69, 2004.

[162] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan, "Anatomy and Performance of SSL Processing," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, March 2005, pp. 197–206.

[163] H. Xie, L. Zhou, and L. Bhuyan, "Architectural Analysis of Cryptographic Applications for Network Processors," in *HPCA-8 Workshop on Network Processors, IEEE Conference on*, 2002.

[164] National Laboratory for Applied Network Research, "NLANR Passive Measurement Analysis." [Online]. Available: http://pma.nlanr.net

[165] CSIX-L1: Common Switch Interface Specification-L1. Optical Internetworking Forum. [Online]. Available: http://www.oiforum.com/public/documents/csixL1.pdf

[166] F. Baker, "Requirements for IP Version 4 Routers," in *RFC 1812*, United States, 1995. [Online]. Available: http://www.ietf.org/rfc/rfc1812.txt

[167] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *Selected Areas in Communications, IEEE Journal on*, vol. 17, no. 6, pp. 1083–1092, Jun 1999.

[168] J. W. Evans and C. Filsfils, *Deploying IP and MPLS QoS for Multiservice Networks: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[169] J. Heinanen and R. Guerin, "A Two Rate Three Color Marker," in *RFC 2698*. United States: RFC Editor, 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2698.txt

[170] J. Adams, "The CAST-128 Encryption Algorithm," in *RFC 2144*. United States: RFC Editor, 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2144

[171] R. Rivest, "The RC4 Encryption Algorithm," *RSA Data Security, Inc.*, 1992.

[172] R. Rivest, "The MD5 Message - Digest Algorithm," in *RFC 1321*. United States: RFC Editor, 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1321.txt

[173] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the h.264/avc standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 17, no. 9, pp. 1103 –1120, sept. 2007.

[174] Cisco Inc., "Cisco ASR 9000 Series Aggregation Services Router," Tech. Rep. [Online]. Available: http://www.cisco.com/en/US/prod/collateral/routers/ps9853/brochure_Cisc%o_ASR_9000_Aggregation_Interactive.pdf

[175] S. B. Wicker, *Reed-Solomon Codes and Their Applications*. Piscataway, NJ, USA: IEEE Press, 1994.

[176] D. Taylor and J. Turner, "ClassBench: A Packet Classification Benchmark," in *INFOCOM 2005. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*.

[177] S. Segars, "The ARM9 Family - High performance Microprocessors for Embedded Applications," in *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, Oct 1998, pp. 230–235.

[178] T. Wolf, "Design and Performance of Scalable High-Performance Programmable Routers," Ph.D. dissertation, Washington Univ., Saint Louis, August 2002.

[179] P.-Y. Chang, M. Evers, and Y. N. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," *International Journal of Parallel Programming*, vol. 25, no. 5, pp. 339–362, 1997.

[180] 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," *Network, IEEE*, vol. 17, no. 6, pp. 6–16, Nov.-Dec. 2003.

[181] S. McCreary and K. Claffy, "Trends in wide area ip traffic patterns: A view from ames internet exchange," *presented at the ITC Specialist Seminar on IP Traffic Modeling, Measurement and Management*, vol. 17, no. 6, pp. 6–16, Sept 2000.

[182] A. Dainotti, A. Pescape, and G. Ventre, "A packet-level characterization of network traffic," in *Computer-Aided Modeling, Analysis and Design of Communication Links and Networks, 2006 11th International Workshop on*, 0-0 2006, pp. 38–45.

[183] K. Pagiamtzisr and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2007.

[184] I. Arsovski, T. Chandler, and A. Sheikholeslami, "A Ternary Content-Addressable Memory (TCAM) based on 4T Static Storage and including a Current-race Sensing Scheme," *Solid-State Circuits, IEEE Journal of*, vol. 38, no. 1, pp. 155–158, Jan 2003.