Copyright

by

Muhammad Faisal Iqbal

2013

The Dissertation Committee for Muhammad Faisal Iqbal
certifies that this is the approved version of the following dissertation:

Workload-Aware Network Processors: Improving Performance While Minimizing Power Consumption

Committee:

Lizy Kurian John, Supervisor

Andreas Gerstlauer

Byeong Kil Lee

James Holt

Earl E. Swartzlander, Jr.

Gustavo de Veciana

Workload-Aware Network Processors: Improving Performance While Minimizing Power Consumption

by

Muhammad Faisal Iqbal, BSEE; MSE

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2013

Dedicated to my family

Acknowledgments

Foremost, I am thankful to my advisor Prof. Lizy John for her excellent guidance throughout my graduate studies, and for all she has taught me. Her caring and cooperative nature made this journey very pleasant. I could not have imagined a better mentor for my PhD study.

I am also grateful to the members of my committee: Prof. Andreas Gerstlauer, Dr. Jim Holt, Dr. Byeong Lee, Prof. Earl Swartzlander, and Prof. Gustavo de Veciana. Their encouragement and insightful feedback helped a great deal in improving the quality of this dissertation. I am also thankful to Prof. Adnan Aziz and Prof Nur Touba who employed me as a Teaching Assistant for various semesters. I am highly indebted to Higher Education Commission of Pakistan who supported the first four years of my graduate studies.

I also thank my fellow group members at the Laboratory for Computer Architecture (LCA). Dimitris Kaseridis, Jian Chen, Ciji Isen, and Umar Farooq helped a great deal when I initially joined the group. Arun Nair was always a great person to talk to regarding research or otherwise. I also enjoyed great discussions over coffee with Jee Ho Ryoo.

I have made some great friends at UT. Aater, Amber, Khubaib, Owais,

Zubair and Ahmed were very generous and helpful when I initially arrived in the US and provided a great support system. I am also thankful to Tauseef, Irfan, Kamil, Rashid, Kamran, Ansab, Ali and Tehmoor for their support in studies and otherwise. I will always cherish the time spent with you guys. I am also thankful to Faisal Mehmood Kashif who has been my motivation for many years. He encouraged me to pursue my higher studies and helped me at each step during this process. I am lucky to know a guy as sincere as him.

I cannot forget the contributions of my high school teachers: Sir Atta, Sir Yaseen, Sir Azhar, Sir Ibrar Hussain, Sir Nadeem, and Sir Ramzan. I am lucky to have these amazing teachers ever as my mentors. They taught me the value of hard work and self belief. They always believed in me and gave me confidence to pursue my dreams.

It is not possible to thank my parents enough for their unconditional love, support and prayers throughout the years. My sisters, Memoona, Sadia and Aamina always supported and encouraged me. My brother, Zirgham, has always been my best friend. I also want to thank him for taking good care of ammi and abbu while I was gone. Finally, I would like to thank my loving wife, Urooj, for believing in me, for sticking with me through all good times and bad, and for never failing to lift my spirits.

Muhammad Faisal Iqbal

The University of Texas at Austin

August 2013

Workload-Aware Network Processors: Improving Performance While Minimizing Power Consumption

Publication No. ____________________

Muhammad Faisal Iqbal, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Lizy Kurian John

Network Processors are multicore processors capable of processing network packets at wire speeds of multi-Gbps. Due to their high performance and programmability, these processors have become the main computing elements in many demanding network processing equipments like enterprise, edge and core routers. With the ever increasing use of the internet, the processing demands of these routers have also increased. As a result, the number and complexity of the cores in network processors have also increased. Hence, efficiently managing these cores has become very challenging. This dissertation

discusses two main issues related to efficient usage of large number of parallel cores in network processors: (1) How to allocate work to the processing cores to optimize performance? (2) How to meet the desired performance requirement power efficiently?

This dissertation presents the design of a hash based scheduler to distribute packets to cores. The scheduler exploits multiple dimensions of locality to improve performance while minimizing out of order delivery of packets. This scheduler is designed to work seamlessly when the number of cores allocated to a service is changed. The design of a resource allocator is also presented which allocates different number of cores to services with changing traffic behavior. To improve the power efficiency, a traffic aware power management scheme is presented which exploits variations in traffic rates to save power. The results of simulation studies are presented to evaluate the proposals using real and synthetic network traces. These experiments show that the proposed packet scheduler can improve performance by as much as 40% by improving locality. It is also observed that traffic variations can be exploited to save significant power by turning off the unused cores or by running them at lower frequencies. Improving performance of the individual cores by careful scheduling also helps to reduce the power consumption because the same amount of work can now be done with fewer cores with improved performance. The proposals made in this dissertation show promising improvements over the previous work. Hashing based schedulers have very low overhead and are very suitable for data rates of 100 Gbps and even beyond.

Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xv**

List of Figures **xvi**

Chapter 1 Introduction **1**

1.1 Network Processor Design Challenges 4

1.1.1 Performance Optimization 4

1.1.2 Power Efficiency 7

1.2 Objectives 9

1.3 Thesis Statement 10

1.4 Contributions 10

1.5 Organization 12

Chapter 2 Background and Related Work **13**

2.1 Introduction 13

2.2 Network Processor Architecture 13
2.3 Performance Optimization in Network Processors 16
2.3.1 Existing Work on Packet Ordering 16
2.3.2 Existing Work on Load Balancing 18
2.3.3 Existing Work on Resource Allocation in Multi-service Routers 20
2.4 Background on Power Management 22
2.4.1 Power Management During Active States (P-states) . . 22
2.4.2 Power Management During Idle States (C-states) . . . 23
2.4.3 Existing Policies for P-state Management 25
2.4.4 Existing Policies for C-state Management 28
2.5 Power Efficiency in Network Processors 30
2.5.1 Power Efficient Architectures 30
2.5.2 Existing Power Management Schemes 31
2.6 Summary 33

Chapter 3 Methodology **34**

3.1 Introduction 34
3.2 Simulation Model 34
3.2.1 Cycle Accurate Simulations 36
3.2.2 Power Modeling 37
3.2.3 Workload Model for Multi-service Routers 37
3.2.4 Parallel vs Pipelined Model 38
3.2.5 Processing Latencies 40

3.3 Benchmarks 42
3.3.1 NpBench 42
3.3.2 PacketBench 44
3.3.3 NetBench 45
3.4 Traffic Traces 45
3.4.1 Real Network Traces 45
3.4.2 Synthetic Traffic Generation 47
3.5 Summary 49

Chapter 4 Load Balancing and Packet Ordering 50

4.1 Introduction 50
4.2 Network Processor Load Balancing 50
4.3 Causes of Load Imbalance 53
4.3.1 Skewed Flow-bundle Sizes 53
4.3.2 Skewed Flow Sizes 54
4.3.3 Burstiness of Internet Traffic 56
4.4 Packet Scheduler Design 57
4.4.1 Load Balancing By Aggressive Flow Migration 57
4.4.2 Aggressive Flow Detection 59
4.4.3 Overhead of the Aggressive Flow Detector 61
4.4.4 Load Imbalance Detection 61
4.5 Evaluation 62
4.5.1 Performance of Aggressive Flow Detector (AFD) 62
4.5.2 Benefit of Limiting Migration to Only Top Flows . . . 66

4.5.3 Analysis of Flows on Migration 68
4.6 Summary 69

Chapter 5 I-Cache Aware Packet Scheduling in Multi-service Routers 72

5.1 Introduction 72
5.2 Scheduling Requirements in Multi-service Routers 73
5.3 Packet Scheduler for Multiservice Routers 75
5.3.1 Monitoring of Processing Requirements 77
5.3.2 Core Allocation Policy 78
5.3.3 Core Release Policy 79
5.3.4 Load Redistribution on Core Allocation 80
5.3.5 Load Redistribution on Core Release 85
5.3.6 Overall Scheme 85
5.3.7 Timing Analysis of LAPS 86
5.4 Evaluation 88
5.4.1 Throughput Improvement with LAPS 88
5.4.2 Overall Performance Improvements with LAPS 90
5.5 Dynamic Behavior of the System 93
5.6 Summary 94

Chapter 6 Efficient Traffic Aware Power Management 96

6.1 Introduction 96
6.2 Motivation 97

6.2.1 The Power Management Problem 97
6.2.2 Power Saving Opportunities 98
6.2.3 Ineffectiveness of Existing Schemes 99
6.3 Traffic Aware Power Management (TAP) 100
6.3.1 Prediction of Computing Requirements 100
6.3.2 Deciding Number of Active Cores 104
6.3.3 Deciding Frequencies of Active Cores 105
6.3.4 Measuring Queue Length 107
6.3.5 Deciding Threshold Values 108
6.4 Evaluation 109
6.4.1 Power Savings with Real Network Traces 109
6.4.2 Effectiveness of DVFS 113
6.4.3 Effectiveness of DES Prediction 115
6.4.4 Packet Queue Behavior 117
6.4.5 Power Saving at Different Traffic Rates 118
6.4.6 Effect of Packet Scheduling on Power Savings 120
6.5 Summary 122

Chapter 7 Conclusions and Future Work 124
7.1 Conclusions 124
7.2 Future Research Directions 126
7.2.1 Thermal Hot Spot Reduction 127
7.2.2 Multiple Scheduling Decisions per Packet 127
7.2.3 Fairness and Quality of Service 128

Bibliography **129**

List of Tables

2.1 Example P-states of a typical processor 23

2.2 Example C-states of a typical processor 24

3.1 Configuration of the processing cores 37

3.2 List of CAIDA traces used in the study 46

3.3 List of Auckland-II traces used in the study 46

4.1 Timing, area and power overhead of aggressive flow detector . 61

5.1 Parameters for synthetic traffic generation 90

5.2 Traces used in experiment for packets of individual services . . 91

5.3 Different traffic scenarios used in experiments 91

6.1 Power management policies implemented for comparison . . . 110

6.2 Comparison of power consumption for different schemes 112

6.3 Average number of active cores 113

6.4 Traffic predictors compared 115

List of Figures

1.1 Internet model showing use of edge and core routers 2

1.2 Variation in traffic at the internet backbone 9

2.1 Architecture of a network processor 14

2.2 Use of P and C states to save power 25

3.1 Simulation infrastructure 35

3.2 Example task graph for an edge router 38

4.1 Packet scheduling in network processors 51

4.2 Effectiveness of hash functions for IP headers 54

4.3 Skewed flow sizes in internet traffic 55

4.4 Load balancer design 59

4.5 Structure of the Aggressive Flow Detector (AFD) 60

4.6 Accuracy of the AFD 63

4.7 Effect of window size on accuracy of the AFD 64

4.8 Effect of sampling on accuracy of the AFD 65

4.9 Packet drop relative Dittman's Load Balancer (DLB) 67

4.10 Number of out of order packets relative to DLB 68
4.11 Number of flow shifts relative to DLB 68
4.12 Number of flows allocated to overloaded core at the time of migration 70
4.13 Number of big flows allocated to overloaded core at the time of migration 70

5.1 Variation in processing requirements in a multi-service router . 74
5.2 Initial assignment of flows by hash function 81
5.3 Flow redistribution after allocation of an additional core . . . 83
5.4 Linear hashing at the end of round 0 84
5.5 Locality Aware Packet Scheduler 87
5.6 Throughput comparison of different schedulers 89
5.7 Comparison of packet dropped under different traffic scenarios 92
5.8 Comparison of i-cache performance under different traffic scenarios 92
5.9 Comparison of packet order under different traffic scenarios . . 93
5.10 Temporal behavior of the resource allocator 95

6.1 Variation in traffic 98
6.2 Power savings on real network traces. 110
6.3 Performance of greedy algorithm that tries to minimize Energy Per Instruction (EPI) 112
6.4 Benefit of combining DVFS with on-off control 114
6.5 Global DVFS vs per core DVFS 115

6.6 Accuracy comparison of traffic predictors 116
6.7 Comparing the power savings of DES predictor with LV predictor 117
6.8 Queue size and filtered queue size 118
6.9 Power consumption comparison at different traffic rates 119
6.10 Combining LAPS and TAP 121
6.11 Power saving benefits of careful packet scheduling 121
6.12 Power saving benefits of careful packet scheduling on a multi-service router 122

Chapter 1

Introduction

Multicore processors have emerged as the mainstream designs in many application domains including general purpose, embedded, networking and graphics processing. From a hardware standpoint, multicore processors are favored by vendors because they facilitate building high performance and high transistor count chips with manageable design complexity by replicating cores. From a software standpoint, applications with sufficient parallelism can have significant performance improvement by running code on multiple cores at the same time. One application domain with abundant parallelism is networking. Networking applications have ample parallelism because multiple packets can be processed by different cores in parallel. This packet level parallelism makes multicore architectures well suited for networking applications.

A *Network Processor* is a special-purpose, programmable device that is optimized for network operations. A network processor is generally a multicore processor that can process network packets at wire-speeds of multi-Gbps.

The main purpose of a network processor is to provide the performance of custom silicon (i.e., ASIC chips) combined with the programmability of general-purpose cores. The ability of a network processor to perform complex and flexible processing and its programmability make it an excellent solution in many demanding network processing environments like enterprise, edge and core routers. Enterprise routers are used within an enterprise and their size depends on the size of the enterprise. More demanding routers are the edge and core routers. Figure 1 shows the use of routers at the edge and core of internet. While the main requirement in core routers is high capacity to handle huge amounts of traffic, edge routers require programmability and flexibility in order to support multiple complex applications like intrusion detection, firewalls, protocol gateways, etc. To meet these requirements, the routers need huge processing power. Network processors have become the main building blocks in these network systems because they provide the required processing power and flexibility.

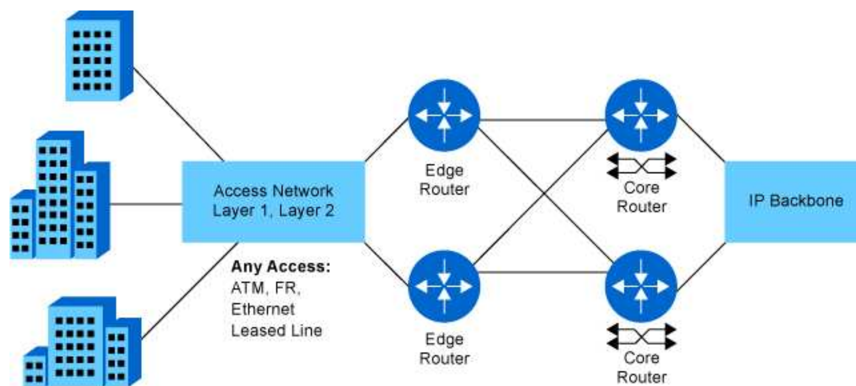


Figure 1.1: Internet model showing use of edge and core routers

A number of network processors have entered the market recently. These processors can be classified into two categories. The first category includes general purpose multicore processors that are adapted to perform networking functions. Examples of such processors are the Sun Niagara [66] and Tilera [29] processors. The second category includes processors which are specifically designed for networking applications. These processors are equipped with hardware accelerators and co-processors in addition to a large number of general-purpose cores. Examples include the Freescale T4240 [6], Broadcom XLP832 [16], EZChip [67], Intel IXP [17] and IBM PowerNP [21]. Both of these categories have a common attribute: they utilize a large number of cores to achieve desirable performance. Network processors with 64 cores or more have been announced by vendors to handle 100 Gbps network speed [13, 2]. With increasing traffic rates and processing demands, the number and complexity of cores in these processors are also on the rise and efficiently managing these cores has become very challenging. The challenges associated with large amount of parallelism are two-fold: (1) how to allocate work to a large number of available cores to maximize performance, and (2) how to achieve the performance goals with minimum power consumption. This dissertation presents schemes to efficiently manage the large number of cores in network processors in order to achieve the desired performance while keeping the power consumption to a minimum.

The challenges associated with network processors are further explained in the next section (Section 1.1). In Section 1.2, objectives of this dissertation

are presented and a formal thesis statement is provided in Section 1.3. Section 1.4 presents the specific contributions made by this dissertation and Section 1.5 describes how the rest of the dissertation is organized.

1.1 Network Processor Design Challenges

1.1.1 Performance Optimization

With ever increasing network traffic rates, multicore architectures for network processors have successfully provided performance improvements through high parallelism. However, naively allocating the network traffic to multiple cores without considering diversified applications and flow locality results in issues such as packet reordering, load imbalance and inefficient cache usage.

Packet Ordering Although the internet is designed to tolerate out-of-order packets, performance of upper layer protocols, such as Transmission Control Protocol (TCP), greatly depends on packet ordering [93]. It is important in applications like Voice Over IP (VOIP) and multimedia transcoding that packets arrive in order because the receiver might not be able to easily reorder the packets. Hence, it is important to preserve the order among the packets of a flow. In this work, a flow is defined as a set of packets that have the same source address, destination address, source port, destination port and protocol. If packets from the same flow are processed by different cores, they can experience different queuing and processing delays, and consequently, the

probability of out of order delivery of packets increases. Careful scheduling of packets is needed in the network processors to minimize out of order departure of packets.

Load Balancing Load balancing is an important technique to efficiently utilize multiple cores in a network processor. Packets arriving at the input should be distributed uniformly to the available processing cores to maximize performance. An unbalanced allocation of load can swamp some cores. As a result, incoming packets assigned to overloaded cores will experience large delays and may even result in packet loss due to limited storage in the network processor.

Data Cache Locality If different cores process packets of the same flow, the data cache will be used inefficiently as the same data is copied to multiple caches. Packet processing needs to access per flow data (state, statistics), as well as more global data (routing table). If packets of a flow always go to the same core, locality can be preserved for both local and global data. Locality in global data comes from the fact that different flows may be hot with respect to different parts of the routing table, i.e., at the lower levels of the tree. The higher levels are hot to all cores. Furthermore, there are many statistics that are kept per flow, per port etc. Each packet may need to update several of these statistics. If multiple cores work on packets of the same flow in parallel, the per flow information needs to be kept consistent across these cores by using synchronization primitives like locks or semaphores. This results in blocking

access and degrades performance. The scheduler needs to account for flow locality to get good performance.

Support for Multiple Services Modern network processors are required to support a rich set of services. For example, a multi-service edge router may need to support encryption, decryption, firewalling, intrusion detection and many other services [65, 107, 55]. The packet processing cores used in these processors are usually small with a small instruction cache (i-cache) of size 8-16KB. These caches can only hold a single program at a time. The performance of a core will deteriorate due to i-cache misses if it has to process packets of different application types. In order to preserve i-cache locality, an efficient resource allocator is needed to divide the pool of processing cores among multiple services. If cores are allocated to services statically at design time based on their worst case requirements, it will result in unnecessary over-provisioning with high system cost. The resource allocator needs to be able to dynamically multiplex cores among services based on runtime traffic requirements in order to keep the processor provisioning level reasonable.

Researchers have proposed using hashing to distribute packets in parallel network processors [33, 64, 101, 102]. The scheduler hashes one or more header fields of the incoming packet and uses the result to decide the target core for that packet. Packets of the same flow are always mapped to the same core because header fields are constant for all packets of a flow. Hence the flow locality and packet order are maintained. Hash based designs are popular because of simplicity, but they do not perform well under highly variable

traffic conditions due to skewed flow sizes or biased hash bundles. Skewed flow sizes refer to the common situation where network traffic constitutes several very high data rate flows and a very large number of low data rate flows [47, 102]. Biased hash bundles mean that internet addresses are unevenly distributed such that the hash function results in uneven allocation of addresses to cores [64]. Furthermore, the hash based schemes need to be adapted for multi-service routers where different packets require different processing and cores are dynamically allocated to services based on traffic variations. Hashing schemes also need to be modified for power saving techniques. These power management techniques power down the underutilized cores when observed network traffic is low [82, 58].

1.1.2 Power Efficiency

Design of power efficient network processors is also very important in order to improve reliability and decrease operational expenditures. The internet infrastructure contributes to approximately 2% of the world's energy consumption [18, 96, 53]. This contribution is likely to increase in the future with exponential growth in number of users and high bandwidth services. According to different studies, routers are major contributors to power consumption in internet infrastructure [22, 53]. The energy consumption in routers is reaching the limits of air cooling [22, 19]. For example, a fully configured Cisco CRS-1 router can consume up to one megawatt of power [19]. A typical router has a set of line cards and each line card has one or more network processors

[11, 3, 81]. The power consumption of a single line card can reach up to 500 Watts [3, 9]. Modern routers can have hundreds of line cards. For example, a Cisco CRS-1 router can house up to 1152 line cards in different chassis. These line cards are densely packed in routers. High power consumption can result in high temperature of parts and failure due to thermal stress. Such failures affect the reliability and availability of networks. This results in lower quality of service and increased expenditures in replacement parts. High power consumption of equipment leads to higher cooling costs and results in increased operational expenditure for the network. According to Ericsson's vice president, "The cost of electricity over the lifetime of network equipment is more than the cost of network equipment itself" [37].

With increasing traffic rate demands and computational complexity, the number and complexity of cores in network processors are on the rise resulting in more and more power consumption. One example of this trend can be seen in Intel's IXP line of network processors. The power consumption has increased from 6 Watts (IXP 1200) to more than 30 Watts (IXP 2800) [17]. Tight power budgets and dense integration requirements call for the design of power efficient network processors.

Multicore packet processing systems are usually designed and provisioned with enough resources to satisfy peak traffic load. But network traffic varies with time and reaches the peak value for only a small portion of time. Figure 1.2 shows traffic observed over two days by the CAIDA monitor [30] at internet backbone in Chicago. There is a huge variation in packet rates and

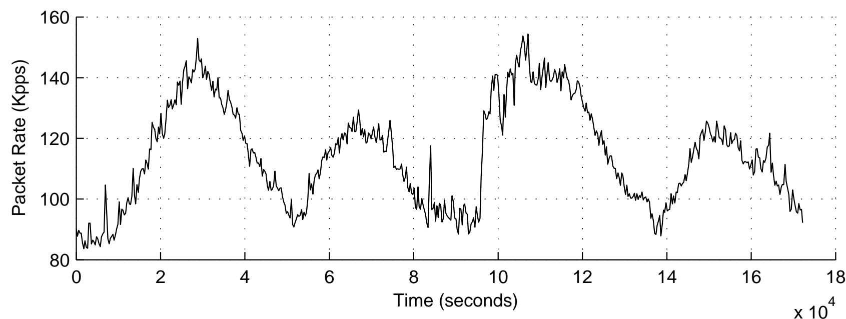


Figure 1.2: Traffic variation over 2 days at the internet backbone in Chicago [30]

thus different processing requirements at different times of the day. Most of the time the traffic rate is below the maximum traffic and processors do not need to run at full capabilities. The low activity periods can be exploited to save power in network processors by running them in low power modes and/or by turning off some processing cores.

1.2 Objectives

The objective of the research presented in this dissertation is to efficiently manage resources of a network processor. The goal is to meet the demands of incoming traffic while minimizing power consumption. The specific objectives are as follows:

1. To design an efficient packet scheduling scheme that provides load balancing, packet ordering, data locality and instruction locality to improve the throughput of a network processor

2. To design a dynamic resource allocation scheme for network processors that allocates resources to services of a multi-service router based on runtime requirements and works seamlessly with the packet scheduler

3. To design an effective power management scheme for network processors that exploits the variability in network traffic over time to save power

1.3 Thesis Statement

A packet scheduler that considers flow and instruction locality improves processing capability (throughput) of a network processor while minimizing power consumption. A dynamic optimization scheme that maximizes per core performance reduces the number of cores needed to handle input traffic and minimizes power consumption.

1.4 Contributions

To address the challenges associated with network processor design, the following innovations are presented.

1. Design of a packet scheduler is presented that achieves load balance with minimum packet reordering. The scheduler uses a hash based design which inherently maintains flow locality and packet order. Load-balancing in case of unbalanced allocation of load to cores is achieved by migrating flows from an overused core to an underutilized core. Flow

migrations are minimized by migrating only aggressive flows. To identify aggressive flows, a novel two-level caching scheme is presented. The scheme is able to identify top aggressive flows with very low overhead and is based on the *annex-cache* idea [62].

2. A core partitioning scheme is presented for multi-service routers to preserve i-cache locality. Each service has exclusive ownership of a subset of cores so that the i-cache locality is maintained. The number of cores allocated to a service changes dynamically with traffic variations. Modifications to the hash based packet scheduler are presented for multi-service routers. These modifications include separate per application map tables and use of incremental hashing to manage scheduling when cores are dynamically allocated to services.

3. A predictive power management scheme for network processors is presented. The scheme uses a low cost traffic and load predictor. The aim is to reduce both active and idle power of cores in the network processors. A new parameter called *traffic_factor* is presented that combines traffic prediction and application processing requirements into a single parameter for efficiently predicting processing requirements.

4. The effectiveness of incremental hashing is demonstrated when the packet scheduling scheme is integrated with the power management scheme that dynamically turns cores on and off to save power. It is further shown that improving per core throughput also helps in conserving energy since

more work can be done by fewer cores if performance of individual cores is optimized by exploiting data and instruction locality.

1.5 Organization

The dissertation is organized as follows:

Chapter 2 presents background and related work. This chapter presents the architecture of a network processor and introduces the basics of power management. Prior work related to performance enhancements and power management of network processors is also presented. Chapter 3 presents the evaluation framework used in this dissertation and explains the set of network traces and benchmarks that were studied. Chapter 4 presents the need to maintain load balancing in network processors in order to improve the throughput and presents techniques to achieve load balancing with minimum overhead. Chapter 5 describes dynamic allocation of resources in a multi-service router. This chapter also explains how dynamic resource allocation is combined with the hash based packet scheduler to efficiently utilize the resources of a network processor. Chapter 6 presents design and evaluation of a power management scheme for network processors. This chapter also presents power improvements when power management scheme is integrated with the packet scheduler of Chapter 5. Finally, Chapter 7 presents the conclusions and points to areas of future work.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter starts with a description of network processor architecture. Next, it presents the prior work done on improving performance of network processors through careful packet scheduling. Then power management is introduced and existing work on power management in microprocessors is discussed. Finally, a summary of power management schemes traditionally applied to network processors is presented.

2.2 Network Processor Architecture

Many architectural variations of network processors exist in the market. Although vendors differ in specific implementations, they share the broad concept of network processors with many cores and accelerators for networking

functions. Architecture for a typical network processor is shown in Figure 2.1. Incoming packets are received by a Frame Manager (FM). FM places the packet payload in a buffer allocated by the Buffer Manager and places the header, a pointer to the buffer and some meta data as command descriptors in the input queues to the processing cores. These general purpose cores process the packets and can offload some of the work to accelerators e.g., some of the work can be placed in the queue for security accelerator (SEC). SEC performs the required processing and puts it back to the return queue. Eventually, the general purpose core sends the packet back to FM via an enqueue after finishing the processing.

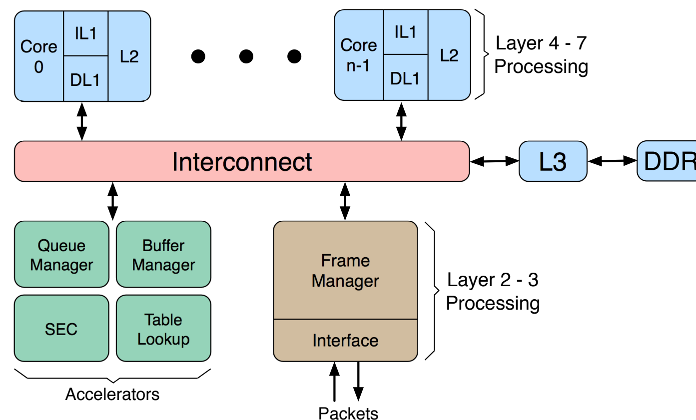


Figure 2.1: Architecture of a network processor

Network processing can be classified as either *Control Plane* or *Data Plane*. *Control Plane* is responsible for control and management processing e.g., maintaining and updating the routing tables. For example, control plane processing may involve executing routing protocols like RIP, OSPF, and BGP,

or control and signalling protocols such as RSVP or LDP. *Data Plane* deals with actual processing involved in packet forwarding. The data plane execution involves compression, encryption, address searches, address prefix matching for forwarding, classification, traffic shaping, network address translation and so on. Traditionally, the general purpose cores in the network processor were responsible for processing both control and data plane packets. However, in modern network processors, control and data plane packets take two different paths. When a packet arrives, a packet classifier in the FM decides whether it is a control plane or a data plane packet. Control plane packets take the *slow path* through the general purpose cores. The data plane packets (Layer 2 or possibly Layer 3) take the *fast path* and are not offloaded to general purpose cores. Fast path processing is handled by the FM itself. The FM is equipped with a large number (32 - 120) of small cores also called I/O Processors (IOP) which are responsible for fast path processing. These IOPs are in-order dual issue cores with non coherent memory, and do not have an operating system. This configuration describes a notional system that represents a class of chips as they look today and moving into the next 3-5 years. In this work, the terms core and IOP are used interchangeably.

Designing a scheduler for the data plane is very challenging. First, the scheduler is in the data path, and therefore, should be as efficient as possible in terms of latency to handle ever increasing traffic rates (100 Gbps and even higher in the future). Second, it should satisfy the requirements of load balancing, flow locality, packet ordering and i-cache locality. Prior work done

on improving the performance of network processors with careful scheduling is presented in Section 2.3. The second challenging aspect of network processor design is to achieve this high packet throughput while maintaining power efficiency. Power management techniques used in microprocessors are introduced in Section 2.4 and some work on applying these power management schemes to network processors is also discussed.

2.3 Performance Optimization in Network Processors

In order to achieve desirable packet processing performance, careful scheduling of packets is required to maintain packet order and exploit data and instruction cache locality. This section presents some prior work done to achieve these goals and explains how this dissertation proposes to overcome their deficiencies.

2.3.1 Existing Work on Packet Ordering

Previous researchers have adopted two different approaches to minimize packet reordering in network processors: order restoration and order preservation.

Order Restoration

This technique allows packets belonging to a flow to be processed out of order by different cores and restores the order at the output [21, 91, 100]. At the input, each packet is tagged with a sequence number and the packets are allowed

to exit the system in strict sequence order. Per flow tagging is needed in order to preserve order among packets of the same flow. This requires keeping per flow information, which is a huge overhead as there can be millions of flows active at a time [78, 79]. Overhead of per flow tagging can be reduced by using global tagging. Global tagging is easy to implement but it is overly restrictive as it forces order even among packets of different flows and results in throughput degradation. Order restoration also requires an expensive synchronization mechanism because multiple cores may be required to update the ordered list of packets of the same flow at the same time. This scheme also results in poor data cache locality because flow locality is not preserved.

Order Preservation

This technique, as the name implies, avoids the overheads of order restoration by preserving packet order.

One example of order preservation is the batch scheduling scheme presented by Guo et al. [46]. In this scheme, a batch of packets is dispatched to cores in a strict round robin manner and is read from cores in the same order. This scheme does not require per flow information but requires expensive synchronization among multiple cores and is not suitable to be implemented for data plane packets. Furthermore, this scheme assumes that each packet requires the same application and does not consider i-cache or flow locality in the algorithm.

Another method to preserve packet order is to use a hash function to

distribute packets to processing cores [33, 64, 101, 102]. The scheduler hashes one or more header fields of the incoming packet and uses the result to decide the target core for that packet. Packets of the same flow are always mapped to the same core because header fields are constant for all packets of a flow. Hence the flow locality and packet order is maintained. Challenges associated with hash function based schemes include quality of the hash function and uneven flow size distribution. Both of these can result in overloading some cores and result in packet loss. In order to avoid the packet loss, a load balancer needs to be integrated with the packet scheduler. This dissertation adopts a hash based scheme for packet scheduling due to its simplicity and ability to preserve order and flow locality and proposes a low cost dynamic load balancing scheme in order to deal with the problem of skewed flow sizes. Some past work related to load balancing in network processors is presented next.

2.3.2 Existing Work on Load Balancing

Dittman presented a hash based packet scheduler and load balancer [33]. When a load imbalance is detected, this scheme migrates arbitrary flows to an under-loaded core. This scheme is referred to Dittman's Load Balancer (DLB) in this dissertation. Such a scheme blindly migrates flows and can result in a large number of flow migrations. A large number of flow migrations results in poor data cache locality and causes many out of order packets.

Shi et al. [102] proposed to only migrate the flows that have high data rates. Since internet traffic has a majority of flows with low activity and a very

small number of flows with high activity, this scheme is able to achieve load balance by migration of a small number of high activity flows with minimized packet ordering. The load balancing scheme presented in this dissertation is based on [102] but the proposed scheme minimizes the overhead of per flow statistics by using a low cost aggressive flow detector. Furthermore, the load-balancer presented in [102] does not consider i-cache locality whereas this dissertation presents a more complete solution that maximizes throughput by considering instruction and data cache localities and minimizes packet reordering. Shi et al. also proposed an adaptive hashing scheme [64] that assures that the weights of the hashing scheme are modified such that the assignment of flow bundles to cores is more evenly balanced for biased hash bundles found in internet traffic. In [101], Shi and Kencl propose to combine the previous two schemes, i.e., adaptive hashing is used in conjunction with the migration of aggressive bundles. This scheme is complementary to the solution proposed in this dissertation and can easily be integrated with the proposed scheduler to further improve the performance of hashing.

The load balancer proposed in this dissertation [57] limits the flow migration only to the aggressive flows. In order to achieve that, efficient identification of aggressive flows is required. Detecting and monitoring aggressive flows is an important part of traffic management and policing. Consequently, there has a plethora of work on how to calculate flow statistics. Initial naive proposals to keep counters for each flow [98, 38] are not scalable when there are millions of flows, which is common in today's network environment. There

have been extensive researches on reducing the overheads of keeping per flow counters [78, 49, 36, 110, 109] to find the accurate estimate of the rates of aggressive flows. In contrast, the proposed packet scheduler in this dissertation merely needs to identify the top aggressive flows without accurately estimating the rates of all flows. The closest related work is done by Yi et al. [79] where a single cache is used to identify "elephant" flows. Experiments done in this dissertation reveal that a single level caching scheme can result in large number of false positives due to many "mice" flows active at any time. This dissertation proposes a novel two-level caching scheme to identify aggressive flows based on annex cache [62]. The proposed aggressive flow detector effectively eliminates the false positives and integrates directly with the scheduler.

2.3.3 Existing Work on Resource Allocation in Multi-service Routers

Another challenge associated with hash based schemes is present in multi-service routers. These routers have to support services like IP forwarding, intrusion detection, IPSEC encryption, IPSEC decryption, etc. Initial proposals of static placement of services on cores [99, 44] do not work well when there is dynamic variation of demands for different services. Many researches have observed the need for dynamic resource allocation in network processors [65, 104, 97]. They observed that if cores are allocated to services statically at design time based on their worst case arrival rates, it will result in unnecessary hardware over-provisioning with high system cost. All services do not

experience their worst case traffic at the same time, so most of the processing resources will be under-utilized. A system that can multiplex cores among different services lowers the total number of cores needed and reduces system cost.

There have been proposals for runtime resource allocations in the past [107, 68], but these schemes consider a packet processing application as a graph where different tasks within the application form the nodes of the graph. These schemes consider adjacency between nodes for task scheduling as packets move between different cores in a pipelined manner during processing. In contrast, this dissertation considers each service as a single entity, i.e., a packet is tied to a single core for the whole processing and graph or pipeline scheduling is not considered (more on this in Section 3.2.4). Wolf et al. [106] observed that the mix of packets destined for each service varies with time. If packets of different services are sent to the same core, i-cache locality cannot be maintained. This results in huge performance overhead. They attempted to address the issue of i-cache locality through careful packet scheduling. When a core becomes idle, their scheme searches for a packet of the same application as the previous one. This searching has a lot of overhead and is not feasible for data plane packets. Although this scheme considers application locality, does not consider data locality and packet order. This dissertation presents a more complete solution to the problem of packet scheduling and resource allocation in multi-service routers in contrast to the prior proposals which have focussed on the individual aspects of the problem.

In addition to achieving good performance, the second aim of this dissertation is to do so with minimum power consumption. Next section presents a brief introduction to power management and presents some work related to power management in network processors.

2.4 Background on Power Management

Modern processors are equipped with capabilities to save power during active periods (P-states) and during idle periods (C-states). P and C-states are part of an open industrial standard called Advanced Configuration and Power Interface (ACPI) [56]. ACPI was proposed by Intel, Microsoft and Toshiba to facilitate the development of Operating System based power management. Policies to manage P and C-states are usually implemented as Operating System modules. Almost all modern operating systems have such modules or governors. This section provides a short background of P and C-states. It also gives an overview of existing policies for C and P-states and explains why these policies may result in un-optimal power management in case of network processors.

2.4.1 Power Management During Active States (P-states)

P-states refer to the different performance states of the processor and provide choices for different power/performance points to adapt to dynamic processing requirements. P-states are an implementation of Dynamic Voltage and

Frequency Scaling (DVFS) and are aimed at reducing dynamic power. Recall that dynamic power is given as

$$P_{dynamic} = k \times v^2 \times f \quad (2.1)$$

where k is a workload and processor dependent parameter determined by switching capacitance and activity of the processor. Dynamic power can be saved if frequency and voltage are reduced. P-states define frequency and voltage levels of the processor so that during times of low processing requirements, frequency and voltage are lowered to save power and energy. P-states are named numerically form P_0 to P_N. P_0 is the highest performance state. Performance and power consumption reduces with increasing P-state numbers. Table 2.1 shows example P-states for a typical processor [25]. These P-states are similar to the P-states of AMD Opteron Processor [1].

P-state	Frequency	Voltage
P0	$F0$	$V0$
P1	$F0 \times 0.85$	$V0 \times 0.96$
P2	$F0 \times 0.75$	$V0 \times 0.90$
P3	$F0 \times 0.65$	$V0 \times 0.85$
P4	$F0 \times 0.50$	$V0 \times 0.80$

Table 2.1: Example P-states of a typical processor

2.4.2 Power Management During Idle States (C-states)

Processor's C-states represent the capability of processor to save power during idle periods. States are named numerically starting from C_0 to C_N, where C_0

represents the active state. As the C-state number increases, the power consumption of the processor decreases and wakeup latency increases. Designers employ different techniques to implement C-states. Low latency techniques include clock and fetch gating whereas high latency techniques include voltage scaling and power gating. Table 2.2 shows an example of C-states. The table shows only three C-states. Modern processors have a large number of C-states. For example, Intel Core 2 Duo has five C-states [105] and some processors even have up to eight C-states [5]. Wakeup latency of processors increases as the

C-state	Response Latency	Relative Power
C0	0	100%
C1	10 μs	40%
C2	100 μs	5%

Table 2.2: Example C-states of a typical processor

processor moves to deeper sleep states. It only makes sense to enter a C-state if inactive time is equal or greater than break even time T_{BE} [23]. T_{BE} for any C-state is defined as the minimum inactivity time required to compensate for the cost of entering that state. The break even time is composed of two terms: the total transition time (i.e., $T_{tr} = T_{enter} + T_{exit}$) and minimum time that has to be spent in that state to compensate for the additional power during transition. If power consumption during transition is less than or equal to on-state power (this is what this study assumes) then

$$T_{BE} = T_{tr} \quad (2.2)$$

This break even time is usually used as a threshold for transitioning into deeper states [23]. For Table 2.2, the break even time will be 20 μs for C1 and 200 μs for C2. Also note that modern operating systems support a tick-less kernel, i.e., idle CPUs do not have to respond to periodic ticks. These CPUs are allowed to remain idle and are woken up by interrupts when a new job arrives for them. Such a tick-less kernel is assumed in this study. Figure 2.2 graphically shows the use of P and C states to save power.

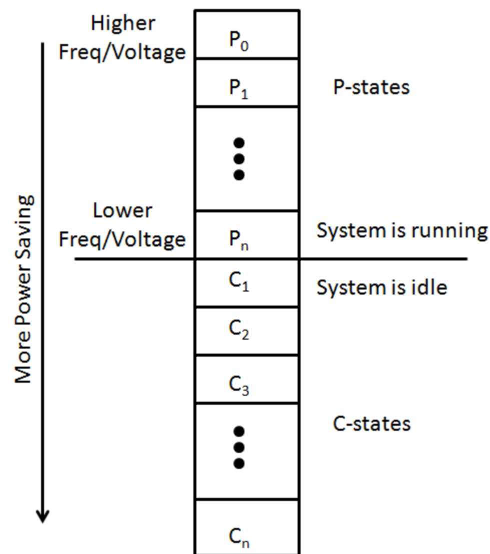


Figure 2.2: Use of P and C states to save power

2.4.3 Existing Policies for P-state Management

Using Processor Utilization

P-state management policies are generally aimed at saving power during performance-insensitive phases of programs. For example, power can be saved during the

memory-bound phase of a program by reducing the clock frequency. Many implementations of these policies use processor utilization to drive the P-states. These policies try to maintain processor utilization within a certain range [25, 20, 50, 12]. Processor utilization or activity level represents the ratio of code execution time (active time) to wall clock time (active + idle time), i.e.,

$$utilization = \frac{Time_{active}}{(Time_{active} + Time_{idle})} \tag{2.3}$$

Listing 2.1: Linux Ondemand frequency Governor

```
1 #define up_threshold 0.90
2 for (each sampling interval){
3   if (utilization > up_threshold)
4      freq = max_freq;
5   else
6      freq = next_lower_freq;
7 }
```

Listing 2.1 shows an implementation of Linux "Ondemand" governor [15]. The algorithm monitors processor utilization for an interval and then makes a decision whether to increase or decrease the frequency. The Ondemand governor is the most aggressive governor in Linux implementations because it tries to settle to lowest frequency in case of zero load and will settle to the highest frequency at peak load. Another relevant governor is a con-

servative governor [15] that is similar to power management module found in Vista [20]. This governor tries to maintain the processor utilization within a range say 0.3 to 0.6.

This type of scheme works well for general purpose applications, which have phases of computations and I/O. Processor utilization is high during compute bound phase and this governor will set the frequency to the highest possible value to finish the work as quickly as possible. During the I/O phases, CPU utilization is not that high and the CPU steps down to the lower frequencies, but in networking applications, it is required that the core is run at just the right frequency to handle the input traffic. The CPU utilization calculated by considering idle time is not very effective since the CPU utilization is itself a function of frequency. Direct information about traffic arrival rates and processing demands are more suitable for networking applications.

Minimizing Energy Per Instruction

Herbert et al. proposed a greedy search method to minimize Energy Per Instruction (EPI) [50, 51] for CMPs. This method is an extension of the technique proposed by Magklis et al. [83]. The P-state controller attempts to operate at a frequency level that minimizes EPI assuming EPI is a bathtub shaped function of voltage and frequency levels. This algorithm assumes the availability of current sensors which help to approximate EPI. After each interval, the controller compares the current EPI with the EPI of previous interval. If EPI is improved, the controller makes a move in the same direction

as the last one. If EPI has increased, it is assumed that controller has overshot the optimal frequency level. It makes a transition in opposite direction as the last one and stays there for N = 5 intervals. After the holding period, the controller continues exploration in a direction opposite to the one that preceded the hold. This type of scheme is un-aware of traffic demands and may result in running at a lower frequency than needed and may result in extra packet loss.

2.4.4 Existing Policies for C-state Management

Using Idle Time

Most implementations of C-state management use *Fixed Timeout* policy. For example, the Ladder governor in Linux is used to implement C-state management [15, 12]. This governor uses elapsed idle time to predict the total duration of the current idle period. When a processor becomes idle, a counter starts. This idle time counter is then compared with pre-defined thresholds. When the counter reaches $C1_{th}$ value, the system is forced into a sleep state C1. The counter continues counting until the processor is woken up by an external event. If the counter reaches $C2_{th}$, the system transitions to C2 and so on. The processor keeps transitioning to deeper C-states until it reaches the lowest power state or it is woken up by an external event. The CPU starts from C1 again when it becomes idle the next time.

As described in Section 2.4.2, the threshold values are decided based on break even times, which are of the order of hundreds of microseconds. This

scheme works well to exploit idle time in general purpose applications e.g., time waiting for user input or response from I/O subsystem where the waiting times are very high, But in case of network processors, the inter-packet arrival times viewed by multiple cores are usually smaller than these thresholds even if the number of active cores is more than the required number to sustain a certain amount of traffic. In other words, this scheme might be too conservative and wastes a lot of power saving opportunities that could be exploited if direct information about traffic and processing demands are available.

Some researchers have proposed more sophisticated approaches that predict idle times based on past idle and active times [32, 24]. These schemes can work better than the pure reactive schemes by applying power saving operations pro-actively, but they still consider C-state management in isolation from P-state management. When both P-states and C-states are available, the idle time becomes dependent on operating frequency of the cores. In such situations, predicting idle periods based on past history of idle periods yields inaccurate results as shown in Section 6.4.

The power management scheme proposed by Simunic et al. [103] is very close to the work presented in this dissertation. The scheme presented by Simunic et al. is for single core portable systems. This scheme uses C-states only during idle traffic times, i.e., when there is zero traffic. The scheme presented in this dissertation is targeted for multicore network processors and is able to change the number of active cores when traffic volume is low.

2.5 Power Efficiency in Network Processors

The need for power efficient internet infrastructure has fueled studies on design of power efficient network processors. A modeling framework for network processors was presented by Crowley et al. [31]. Franklin et al. also developed an analytical model to explore design space for power efficient network processors [41], but they do not study any specific low power techniques for network processors. This section presents low power techniques that have been applied to network processors and also explains power management techniques that are traditionally applied in network processors.

2.5.1 Power Efficient Architectures

Several researchers have worked on designing power efficient architectures for network processors. Memik et al. [87] observed that reduction in bus activity can help save significant power. They proposed a data filtering technique to reduce bus accesses. They demonstrate that in networking applications most of the L1 data cache misses are caused by only a few instructions. Their proposed technique uses a locality prediction table to detect load accesses with low temporal locality and a data filtering engine that processes the code segments surrounding the low temporal accesses. Zane et al. [111] and Kaxiras et al. [63] propose power efficient TCAM structures to be used in packet processors. Mallik et al. [85] study the relationship between lowering the voltage for the cache memories and transient errors. In [70] Lee argues that if static instruction scheduling is used, the network processor can get higher

performance with tremendous advantages in power, since it does not have instruction window wakeup/select logic, reorder buffer, and other scheduling related hardware modules. In similar spirit, Bengu et al. propose to use a computation reuse cache to reduce energy in network processors [72]. All these schemes can be applied in a complementary manner to the proposals made in this dissertation.

2.5.2 Existing Power Management Schemes

Kuang et al. [69] proposed to use DVFS for pipelined networking applications in network processors. They assumed that each stage of the pipeline is allocated to a different core and argued that the cores processing non-critical stages of the pipeline do not need to operate at full speed. They allocate lower frequencies to non critical stages statically and do not consider traffic variations for frequency scaling. In contrast, the power management scheme presented in this dissertation is aimed at exploiting traffic variations to save power. Furthermore, this dissertation considers packet processing as a single application and is not aimed at the pipelined model.

Luo et al. [80] used CPU utilization to decide frequencies of cores. All cores are active all the time in this scheme. In another study, Luo et al. [81, 82] proposed to use clock gating to change number of active cores. This scheme was targeted for a specific architecture and thread model of a network processor where each thread goes to a thread queue after finishing processing. On arrival of a new packet, a thread from the thread queue is woken up to

process the packet. They used number of idle threads in the thread queue to decide whether to clock gate a core or not. Using the number of idle threads works fine if each processor is assumed to run at maximum frequency, but if each core is allowed to change its frequency, the number of idle threads does not effectively represent the load. For example, consider a situation where two processors are active and running at half of the maximum frequency. These processors will be utilized 100% of the time to handle a traffic that a single processor could handle at full speed, but the threads running on slow cores will never enter the thread queue and no cores will be turned off. Hence the number of idle threads does not represent processing requirements in this situation. Furthermore, this scheme is also a reactive scheme and uses only clock gating. The transition overheads for clock gating are small so the reactive scheme works well. Since this scheme is targeted for 280 nm, leakage power is not a big issue and clock gating works fine, but in modern technologies, the power consumption is usually dominated by leakage power and hence it is important to utilize deep sleep states, which have additional power savings even beyond DVFS and clock gating. These deep sleep states have high overheads in terms of transition delays and the power management scheme should be aware of these delays. In contrast, the proposed power management scheme [58, 60] considers multiple sleep states (C-states) and also makes use of P-states to save power of the active cores.

Another set of studies that can be used to complement the proposed scheme targets various parts of internet infrastructure for power efficiency.

Chiaraviglio et al. [28] present a scheme to turn off links and nodes during periods of low activity while still guaranteeing full connectivity. Nedevschi et al. [90] show that in addition to putting elements to sleep, link rate adaptation with varying traffic can help further to save power.

2.6 Summary

This chapter has presented some background information on issues of packet scheduling and power management in network processors. Some previous work related to these issues is also surveyed. The next chapter will discuss the methodology used to evaluate the proposals of this dissertation and subsequent chapters present specific proposals and their evaluation.

Chapter 3

Methodology

3.1 Introduction

This dissertation uses simulations to evaluate the proposed optimization techniques for network processors. Although each proposed scheme requires modifications to the original simulation infrastructure, the basic methodology remains the same. This chapter provides a detailed description of the various tools and benchmarks used in this dissertation.

3.2 Simulation Model

For evaluating different techniques, a simulation model in SpecC [43] is developed. SpecC is similar to systemC [45] in its design and philosophy. The simulation model is inspired from the host compile simulation approach used in the SCE simulator from UC Irvine [34]. This approach pairs a high level

functional model with back annotation of statically determined timing and power estimates in order to achieve fast and accurate simulations. Following a similar approach, the SpecC model used in this dissertation models the network processor at a very high level while the timing and power estimates for packet processing applications are back annotated to this model.

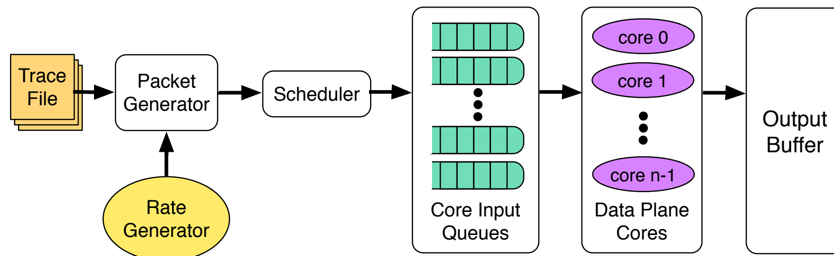


Figure 3.1: Simulation infrastructure

Different components of the simulator are shown in Figure 3.1. The packet generator module creates the input traffic for the network processor. It uses both real traces and a synthetic traffic rate governor. Details of the traffic traces are provided in Section 3.4. The packet scheduler module implements the different scheduling schemes discussed in the dissertation and chooses an input queue for an incoming packet. The queueing system assumes there is a single input queue associated with each core. The size of the queues is an important parameter in network processor configurations. A very large input queue can help in reducing packet drops but it increases the response time. Previous research has studied the effect of bounded queueing on drop rate and response time and observed that a moderate queue size of around 100 entries results in good response time while keeping the drop rate low [42, 77]. The

simulations carried out in this dissertation use a queue size of 100 packets unless otherwise specified.

Each core extracts packets from the input queue, performs the packet processing functions and enqueues the packet into an output buffer. The processing latencies for packet processing are taken from cycle accurate GEMS simulations and McPAT power model. Details of cycle accurate simulations and benchmarks used in this study are provided in the coming sections.

3.2.1 Cycle Accurate Simulations

In order to study detailed characteristics of network processing benchmarks, this dissertation uses the multi-facet execution-driven multiprocessor simulator (GEMS) tool set from the University of Wisconsin [86]. GEMS is combined with Simics [84] to provide a detailed, cycle-accurate simulator by decoupling the simulation functionality from the timing. Simics provides a robust environment to boot the unmodified OS along with the functional simulator. GEMS timing modules interact with Simics to determine when Simics should execute an instruction. However, the result of execution of instruction is still dependent on Simics. Therefore, the two tools work in lock-step mode. Even though, GEMS decouples the functional simulation and the timing simulation, the functional simulation is still affected by the timing simulator, allowing the system to capture timing-dependent effects. This tool set is used to model the multicore processors. The parameters of the individual cores are listed in Table 3.1.

Frequency	Pipeline	Branch Predictor	I-Cache	D-Cache
1GHz	7 stage 2-issue	gshare/BTB 128 entry each	16KB 2 way	32KB 4 way

Table 3.1: Configuration of the processing cores

3.2.2 Power Modeling

For power modeling, the Multicore Power, Area and Timing (McPAT) tool from HP Labs is used [73]. McPAT has quickly become a power modeling standard in both academia and industry. It is an integrated power, area and timing modeling framework for multi-threaded and multicore architectures. It models power, area and timing simultaneously and supports comprehensive early stage design space exploration for multicore and many core processor configurations ranging from 90nm to 22nm and beyond. McPAT includes models for a complete chip multiprocessor, including in-order and out-of-order cores, network on chip, shared caches, and integrated memory controllers. McPAT models timing, area and dynamic, short-circuit and leakage power for each of the device types forecast in the ITRS road map including bulk CMOS, SOI, and double-gate transistors. McPAT has a flexible XML interface that makes it ideal to be used with any performance simulator.

3.2.3 Workload Model for Multi-service Routers

A multi-service router needs to handle packets with diverse processing requirements. Different processing requirements of packets are depicted in Figure 3.2. The workload graph of Figure 3.2 is based on methodology presented by

Huang et al. [55] and represents typical applications needed on an edge router. The incoming packets can be serviced by one of the four services represented by different paths of Figure 3.2. *Path 1* describes the processing needed by outgoing packets that are tunneled via VPN. *Path 2* represents the default handling of packets. *Path 3* is the path of incoming packets on edge router that are scanned for malware and *Path 4* is for incoming VPN packets that are decrypted and scanned for malware.

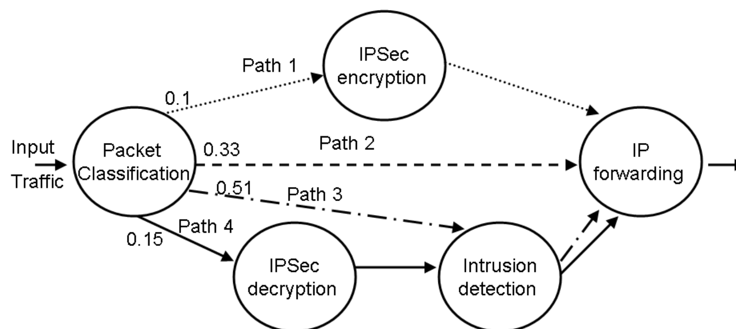


Figure 3.2: Example task graph for an edge router

3.2.4 Parallel vs Pipelined Model

There are two possible choices for implementing such a workload on network processors: a) a pipelined model; b) a run-to-completion (or parallel) model. In a pipelined model, the processing needed by a packet is partitioned into multiple sequential stages. Each stage is than mapped onto one or more cores. In a run-to-completion model, the application is not subdivided and the packet is tied to a single core for life time of its processing. Researchers have shown that the run-to-completion model results in superior performance as compared

to the pipelined model because of its simplicity and low overhead [21, 100]. Hence, the parallel model is adopted in this dissertation. A few advantages of the parallel model are listed below.

Low Communication Cost

In the parallel model, a packet is tied to a single core for its whole processing whereas high communication cost is involved in the pipelined model where the packet has to move between different cores for processing of individual pipeline stages.

Programmability

Parallel models are easier to program because the users can replicate the same code on each core. For the pipelined model, the programmer needs to divide the application carefully into balanced stages. This cannot be easily done because some tasks such as pattern matching cannot be further subdivided [100]. In case of a change in protocol or application implementation, it is easy to update the code for the parallel model whereas such a change requires repartitioning of tasks in the pipelined model. The change in processing of a single stage not only affects itself, but also requires the code to be repartitioned and remapped on all stages to get the maximum performance.

Scalability

The parallel model is very scalable because the same code is replicated across cores to improve performance whereas to improve performance of the pipelined model, one has to increase the pipeline stages and repartition the task. Repartitioning of tasks is not always feasible.

Power Management

The parallel model easily integrates with the power management schemes because individual cores can be turned off without affecting other cores. Such power management schemes are difficult to implement in a pipelined model where all stages are necessary for processing of a packet.

Robustness

The parallel model can easily and gracefully handle failure of individual cores, whereas greater effort is needed in the pipelined model to handle such failures.

3.2.5 Processing Latencies

Each packet of a service i, experiences a Processing Delay (PD_i) in the core based on the following equation

$$PD_i = T_{proc,i} + FM_{penalty} + CC_{penalty} \tag{3.1}$$

Where $T_{proc,i}$ is the processing time, $FM_{penalty}$ is the penalty due to flow migration and $CC_{penalty}$ is the cold cache penalty which occurs when a subsequent packet needs different processing than the previous packet. $T_{proc,i}$ is derived from real delays seen by the packets when the packet processing is implemented in software on a full system GEMS [86] simulator. The configuration of in-order cores is shown in Table 3.1. The packet processing applications are executed on the GEMS simulator and a packet processing delay model for each service is derived. T_{Proc} is measured to be 0.5μs for path 2, i.e., IP forwarding. For path 3, it is measured to be 3.53μs. For Path 1, it also depends on the packet size and is given as

$$T_{proc,path1} = 3.7\mu s + \frac{PacketSize}{64byte} \times 0.23\mu s \tag{3.2}$$

Similarly the processing time for path 4 is given as

$$T_{proc,path4} = 5.8\mu s + \frac{PacketSize}{64byte} \times 0.21\mu s \tag{3.3}$$

$FM_{penalty}$ is set to four cache misses (0.8μs) conservatively. Note that this is just a conservative estimate. In reality, a flow migration can cause a lot more misses since studies have shown that only per flow data amounts to several hundred bytes [75] and there can be additional misses due to route table look up as well [48]. Because of small the i-cache, these cores can hold instructions of only the last executed program (e.g., the AES encryption used in IPSEC requires 16KB). So whenever a packet of a different service arrives at a core,

it will experience a cold cache penalty. The cold cache penalty is set to 10μs which is the cold cache penalty for the smallest service, i.e., IP Forwarding. In practice this penalty will be higher because many services are larger and a context switch can result in some d-cache misses too. For simplicity, these data misses due to context switch are ignored in this work.

3.3 Benchmarks

Several packet processing applications are chosen from NpBench [71], PacketBench [95] and NetBench [88]. These applications include both control plane and data plane benchmarks.

3.3.1 NpBench

FRAG

FRAG is a packet fragmentation application. IP packets are split into multiple fragments if packet size is greater than the Maximum Transmission Unit (MTU). Header fields have to be adjusted and a header checksum needs to be computed for each fragment.

MPLS

MPLS is a forwarding technology, which avoids the overhead of the lookup of bulky headers and uses short labels for forwarding at the edge of MPLS domain. The NpBench version of this application concentrates on two control

plane aspects of MPLS: label generation and label distribution.

SSLD

The SSL Dispatcher is one example of a content-based switching mechanism. SSL typically runs over TCP, which is used for secure processing of e-commerce applications. Once a TCP connection is established, the client and the server authenticate each other and exchange a session key. This phase is known as the SSL handshake and is a computationally heavy workloads as it typically involves public key cryptography. Based on the session ID, it decides which server node has the session state corresponding to this session. The SSL Dispatcher maintains the session ID information, sharing the SSL information among the nodes in the cluster. When reconnecting to the same server, a client can reuse the session state established during a previous handshake.

WFQ

WFQ is a queue scheduling algorithm to serve packets in order of their finish times considering the weight on connections. Various lengths of packets from incoming traffic are classified into different queues, which can be used for differential service. WFQ bases its scheduling decisions based on a packet's estimated finish time.

3.3.2 PacketBench

IPV4R

This is the IPV4-radix application from PacketBench. It performs RFC1812-compliant packet forwarding and uses a radix tree structure to store entries of the routing table. The routing table is accessed to find the interface to which packet is sent, depending on its destination IP address. The radix tree data structure is based on an implementation in the BSD operating system.

IPV4T

This is the IPV4-trie benchmark and is similar to IPV4-radix benchmark. It also performs RFC1812-based packet forwarding. This implementation is derived from an implementation for the Intel IXP processor. This application uses a multi-bit trie data structure to store the routing table, which is more efficient in terms of storage space and lookup efficiency.

IPSEC

IPSEC is an implementation of IP Security Protocol, where the packet payload is encrypted using the Rijndael Advanced Encryption Standard (AES) algorithm.

3.3.3 NetBench

IPCHAINS

This is a firewall application that checks the IP source of each coming packet and decides either to pass the packet through the firewall (accept), to deny the packet (deny), to modify it (masq), or to reject the packet and send information to the sender (reject).

3.4 Traffic Traces

The proposals made in this dissertation are evaluated using both real and synthetic network traces. The details of these traces are listed below.

3.4.1 Real Network Traces

For this work, real network traces were taken from two sources:

CAIDA Traces

This data set contains traffic traces from CAIDA's equinix-chicago and equinix-sanjose monitors on high speed internet backbone traffic links [30]. Both these monitors are connected to OC192 links. The traces in this data set are of 1 minute long duration and were captured in the year 2011. CAIDA also makes real time traffic reports available on its website. The details of traces used in this study are listed in Table 3.2. This dataset contains anonymized traffic traces from CAIDA's equinix-sanjose monitor [30]. This monitor is connected

to OC-192 link. These set of traces were captured in the year 2011 and are of duration of 1 minute each.

Trace	Name
Caida 1	20110120-125905.UTC.anon.pcap.gz
Caida 2	20110120-130000.UTC.anon.pcap.gz
Caida 3	20110120-130100.UTC.anon.pcap.gz
Caida 4	20110120-130200.UTC.anon.pcap.gz

Table 3.2: List of CAIDA traces used in the study

University of Auckland Traces

This set of traces, also known as AUCK-II, was captured at University of Auckland and captures the traffic between the university and its ISP [14]. All connections from the university to external world pass through this measurement point. Most traces are of 24 hour duration with some being shorter. All non-IP traffic has been discarded and only TCP, UDP and ICMP traffic is available in these traces.

Trace	Name
Auckland 1	20000614-181539-0.gz
Auckland 2	20000614-181539-1.gz
Auckland 3	20000619-183717-1.gz
Auckland 4	20000621-105006-0.gz
Auckland 5	20000621-105006-1.gz
Auckland 6	20000630-175712-0.gz
Auckland 7	20000630-175712-1.gz
Auckland 8	20000703-152100-0.gz

Table 3.3: List of Auckland-II traces used in the study

3.4.2 Synthetic Traffic Generation

In order to study the behavior of different traffic scenarios, synthetic traces were also generated. In order to keep these traces as realistic as possible, the synthetic trace generation mechanism only governs the traffic rates, the packet header and payloads are still taken from the real network traces of Section 3.4.1.

The synthetic trace generation methodology is based on *Holt-Winters* forecasting as applied to networking in [26] and [55]. This method decomposes a time series into three components: a base line component a_t, a linear trend component b_t and a seasonal trend c_t. The prediction for the time interval is given as

$$\hat{y}_{t+1} = a_t + b_t + c_{t+1-m} \tag{3.4}$$

where t is in index denoting the time period and m is the period of the seasonal cycle. The individual components of this equation are calculated using exponential smoothing. The baseline is calculated as

$$a_t = \alpha(y_t - c_{t-m}) + (1 - \alpha)(a_{t-1} + b_{t-1}) \tag{3.5}$$

The linear trend or slope is calculated by the equation

$$b_t = \beta(a_t - a_{t-1}) + (1 - \beta)b_{t-1} \tag{3.6}$$

The equation for the seasonal component is

$$c_t = \gamma(y_t - a_t) + (1-\gamma)c_{t-m} \tag{3.7}$$

where α, β and γ are parameters for baseline, trend and seasonal components respectively and are estimated in such a way that the MSE of the prediction is minimized. In this work, y_t means the number of packets arriving in interval started at time t.

By applying this concept, the incoming traffic rate can be governed by the equation,

$$y_t = a + b_t + c \times S_{(t\%m)} + N_\sigma \tag{3.8}$$

where a, b, c and m are the programmable parameters for baseline, linear trend and seasonality. The shape function S was introduced in [55] to allow any shape function to be used for periodic component and is normalized to produce values between -1 and 1. The parameter c determines the amplitude of the seasonal component. The noise function N adds random noise to the traffic rate with mean zero and parameterized by standard deviation σ, i.e., this σ determines how much noise is superimposed onto the traffic. It is expected that traffic with a small baseline will have a variation that is small, hence σ is set proportional to baseline a traffic.

In order to generate traffic for n different services in a multi-service router, n traffic generation functions $y1_t$, $y2_t$, ..., yn_t are used, each with its own set of parameters. Total incoming traffic in this scenario is the sum of

incoming traffic for each service, i.e.,

$$Y_t = \sum y_{it} \tag{3.9}$$

3.5 Summary

This chapter has presented the evaluation methodology used in this dissertation. It explains the simulation framework, benchmarks and traffic traces used. In the coming chapters, the proposed techniques and their evaluations are presented.

Chapter 4

Load Balancing and Packet Ordering

4.1 Introduction

This chapter presents a design of a load balancer for network processors. The design goal is to achieve load balance while minimizing packet reordering. The effectiveness of the design is demonstrated through simulations using real network traces.

4.2 Network Processor Load Balancing

Load balancing is an important technique to efficiently utilize multiple cores in a network processor. An unbalanced allocation of load can swamp some cores. As a result, incoming packets assigned to overused cores will experience

larger delays and may result in packet loss due to limited storage in a network processor. In addition, since the majority of the internet traffic is based on TCP [30], care must be taken to avoid out of order departure of packets within a flow. Although the internet is designed to tolerate out of order packets, performance of upper layer protocols such as TCP greatly depends on packet ordering because out of order packets can falsely trigger congestion control mechanisms and degrade the throughput unnecessarily [93]. Furthermore, packets from the same flow need to be sent to the same core in order to exploit data cache locality for flow state and routing table information.

In a network processor, load distribution among the multiple cores is performed by a packet scheduler. Figure 4.1 shows a generic diagram of a packet scheduler. A packet scheduler receives an incoming packet form high speed link with traffic rate λ and forwards it to one of the cores for processing. Each cores processing power is μ_i and the total processing power of the network processor is $\mu = \sum \mu_i$. This chapter focuses on the scheduler for data plane

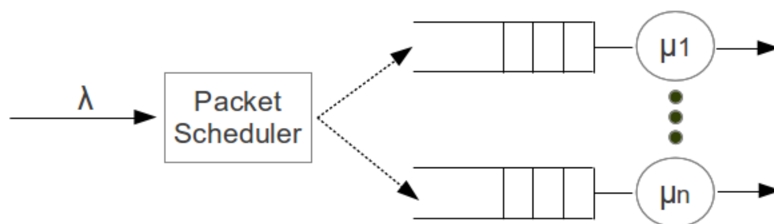


Figure 4.1: Packet scheduling in network processors

packets and does not consider control plane packets. The design of a scheduler

for data plane is particularly challenging. First, the scheduler is in the data path, and therefore, should be as efficient as possible in terms of latency to handle ever increasing traffic rates (100 Gbps and even higher in the future). Second, it should satisfy the requirements of load balancing, packet ordering, data cache and instruction cache locality[1].

Previous researchers have presented many load-balancing schemes [46, 100, 91]. These schemes can be classified into two categories:

1. **Packet Level Load Balancing:** These schemes schedule each packet independently to achieve uniformity in load assignment. For example packets may be distributed in a round robin fashion [46], or an incoming packet is allocated to the least loaded core [100]. These schemes have two drawbacks. First, these schemes reorder packets very frequently. Second, these schemes cannot utilize the data cache efficiently because they send packets belonging to the same flow to different cores.

2. **Flow Level Load Balancing:** Flow level schemes generally use hashing to distribute flows to individual cores [33, 64, 101, 102]. The scheduler hashes one or more header fields of the incoming packet and uses the result to decide the target core for that packet. Packets of the same flow are always mapped to the same core since header fields are constant for all packets of a flow. Hence the flow locality and packet order is maintained.

[1]This chapter does not consider i-cache locality. The scheduler presented in this chapter is extended in Chapter 5 to make it i-cache aware for multi-service routers.

The scheduler presented in this chapter uses a hash-based approach because of its simplicity and obvious advantage of packet ordering and flow locality. This chapter discusses the challenges associated with a hash based packet scheduler and presents the design of a scheduler that overcomes these challenges.

4.3 Causes of Load Imbalance

Hash based designs are popular choices due to their low overhead. These designs only need to compute a hash function to get the target core for a packet, but there are several dynamic properties of network traffic that make load balancing task challenging for hash-based designs.

4.3.1 Skewed Flow-bundle Sizes

The quality of hash function plays an important role in distributing the flows evenly to all processing cores. All the flows that map to the same core or bin in the map table are referred to as a flow-bundle. Uniformity of flow bundle sizes means that the hash function has distributed flows very effectively to the processing cores. Under ideal conditions, each flow bundle should have a size of $\frac{F}{M}$. Where F is the number total number of flows seen in time T and M is the number of bins or cores.

Many researchers have worked on designing effective hash functions for internet addresses [27, 52, 101]. It has been shown that CRC16 performs very

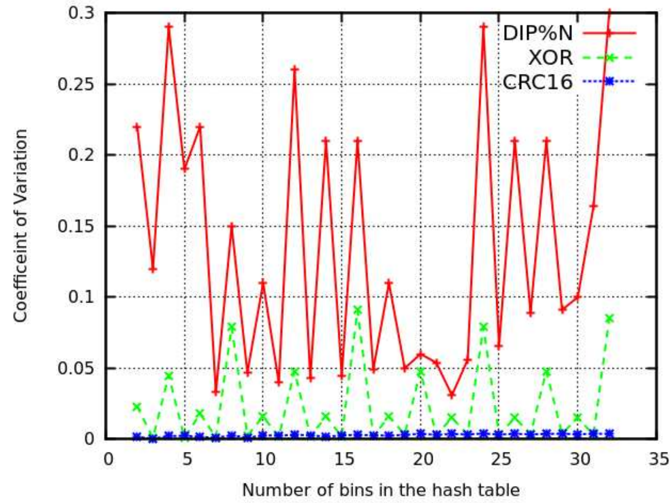


Figure 4.2: Coefficient of variation of number of flows allocated to bins by different hash functions

well for internet traffic [27]. Similar results are observed in this study. Figure 4.2 shows performance of different hash functions for internet backbone trace Caida 1 from Table 3.2. The y-axis shows the coefficient of variation (i.e., ratio of standard deviation to mean) of the number of flows mapped to each bin when the number of bins are varied from 2 to 32. It can be seen that both XOR and CRC16 hash functions perform very well for this trace. Comparable results were observed for all other traces as well.

4.3.2 Skewed Flow Sizes

Even with perfect distribution of flows to cores, load imbalance can still occur since all of the flows are not the same size. In fact, it is well known that network traffic constitutes only few heavy-hitter (high data rate) flows and many low data rate flows [47, 102]. Figure 4.3 demonstrates this behavior in

real network traffic. The plot shows the popularity of flows (y-axis) with most popular flow plotted first (x-axis).

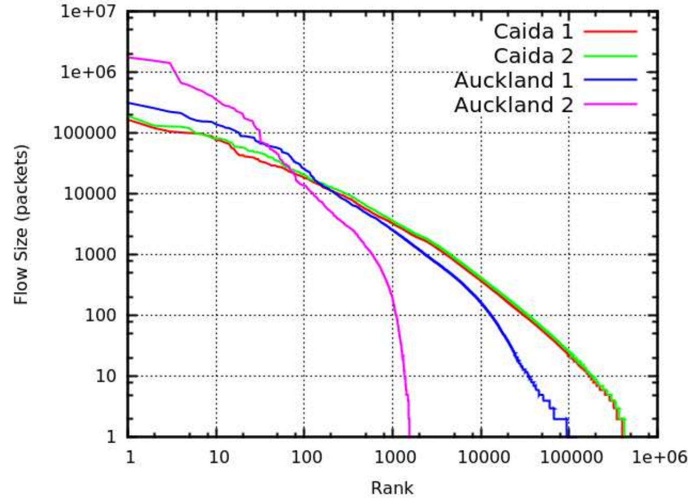


Figure 4.3: Distribution of flow sizes in real network traces. Rank 1 is the flow with the highest flow size.

Shi et al. have shown that hashing alone cannot balance the load under this highly skewed distribution of flow sizes [102] and can result in overloading some cores. In this scenario, the load on each core should be monitored and adjusted dynamically to migrate some load to underutilized cores. Care must be taken because it is desirable to minimize the number of flow migrations. Flow migrations result in out-of-order packets and also badly affect data cache performance.

In order to minimize the number of flow migrations, previous research has made the observation that migrations should be limited to only top aggressive flows [102]. In this way load balance can be achieved by minimum flow migrations. However, previous research kept per flow statistics to identify

aggressive flows. Maintaining per flow statistics has a lot of overhead and is not possible in realistic designs. Although many per flow statistics are maintained by software, accessing those software statistics is very time consuming for a scheduler that is trying to schedule data plane packets. The data plane packet scheduler needs to function with minimum software intervention for good performance.

This research presents the design of a hardware scheduler for data plane packets. A novel low-overhead hardware technique to identify aggressive flows is presented. The aggressive flow detection scheme is based on the two-level caching idea of annex-cache [62] used in general purpose applications. The caching based aggressive flow detector integrates readily with a hash based packet scheduler. The complete design and evaluation of the scheduler is presented in this chapter.

4.3.3 Burstiness of Internet Traffic

It is well known that packets of a flow travel in groups or trains [61]. If a large number of packets of the same flow arrive in a small interval of time, they can overload the core. This can result in a temporary load imbalance. A common technique to deal with bursty traffic is buffering. The dynamic load balancing scheme presented in this chapter also deals with this problem, by relocating the flows in case the burst size exceeds the buffer size.

4.4 Packet Scheduler Design

The design goals for packet scheduler are: a) To achieve high throughput by maintaining flow locality. b) To minimize out of order departure of packets. c) To have a low overhead to sustain high traffic rate.

The proposed packet scheduler uses a hash based design which is a natural way of maintaining flow locality and packet order. The scheduler is called Locality Aware Packet Scheduler (LAPS). When a packet arrives, its flow identifier is extracted from the header. The flow identifier is a five tuple consisting of source and destination IP addresses, source and destination ports and protocol ID. This five tuple is hashed using CRC16 to get an index into a map table. CRC16 is shown to provide good performance for hashing IP headers [27]. The map table[2] stores target core ID where the packet is eventually forwarded. In the presence of skewed flow size distribution as shown in Figure 4.3, the scheduler identifies and migrates the aggressive flows from the overloaded core to achieve load balance. An efficient scheme for identifying and migrating aggressive flows is presented.

4.4.1 Load Balancing By Aggressive Flow Migration

When a core becomes overloaded, i.e., its queue size reaches a threshold, the scheduler needs to migrate some of the incoming traffic from that core to a less loaded core. This migration of flows has two drawbacks: One, it makes

[2]Map table is used instead of direct hashing because it allows dynamic adaptations presented in the next chapter.

some cached data in the source core useless and triggers some cold misses in the cache of newly allocated core. Two, flow migration makes it harder to maintain the order among packets of the flow. The new incoming packets will potentially experience less queuing delay as compared to older packets that are waiting in the overloaded core's queue.

To avoid the above two situations, it is desirable to minimize the number of flow migrations. If only the most aggressive flows can be identified and migrated, load balance can be achieved with minimum disruption, i.e., only a few flows need to be migrated to achieve load balance. In order to achieve this, a low cost mechanism is need to identify top aggressive flows. This research proposes a novel cache based hardware called Aggressive Flow Detector (AFD) to identify the top flows. The hardware consists of a small fully associative cache called Aggressive Flow Cache (AFC). the AFC is augmented with a cache assist called annex cache. Detailed architectures of the annex cache and the AFC are presented in Section 4.5.1.

Figure 4.4 presents the scheduler design. The incoming packets are hashed to get an index into a map table that stores the target core IDs. On load imbalance, the incoming packet flow to the overloaded core is migrated to the least loaded core if the flow is identified as an aggressive flow by the AFD. The decision is recorded in the Migration table. So the future packets of the same flow are always migrated to the newly allocated core. The scheduler gives priority to the output of migration table over the default hash table. If the input queue indicated by the scheduler is filled up, the incoming packet is

dropped.

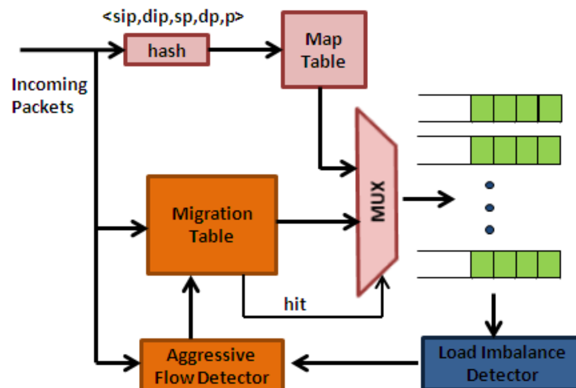


Figure 4.4: Load balancer design

4.4.2 Aggressive Flow Detection

The design of Aggressive Flow Detector is based on *annex cache*. Annex Cache was proposed by John [62] to exploit locality in the memory references in general purpose processor workloads. This study shows that such a structure can be very useful in to identify aggressive flows.

The AFD has two main components as shown in Figure 4.5. One component is a small fully associative cache called the Aggressive Flow Cache (AFC). The AFC holds the IDs of the top aggressive flows. All entries into the AFC come via the annex cache. Items referenced only rarely will be filtered out by the annex cache and will never enter the AFC. The basic premise is that a flow deserves to enter the AFC only if it proves its right to be in the AFC by showing locality in the annex cache. The annex cache also serves as a victim cache and provides some inertia before a flow is excluded from the

AFD. Both the AFC and the annex cache use Least Frequently Used (LFU) replacement policy.

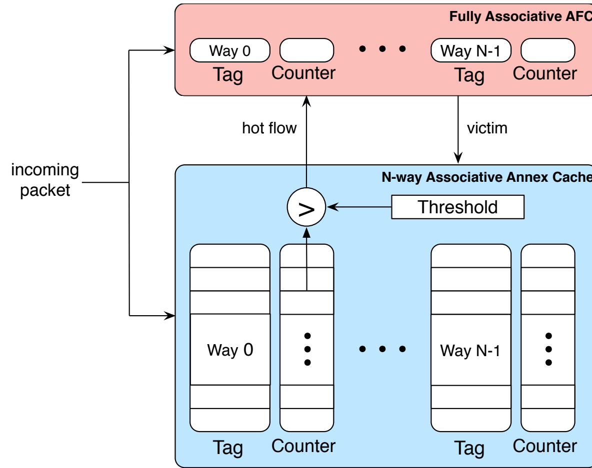


Figure 4.5: Structure of Aggressive Flow Detector (AFD)

The design of the AFD is slightly different from the one presented in [62] because in the AFD the annex cache is bigger than the AFC. A larger AFC is undesirable because the proposed scheme wants to limit the number of monitored aggressive flows. The annex cache is a bigger structure that serves as a qualifying station for large number of flows to demonstrate their eligibility to be cached into the AFC. When a packet arrives, its flow ID is checked in both the AFC and the annex cache. If it is a hit in the AFC, the hit counter is incremented. On a hit in the annex cache, the flow counter is incremented and the value is compared with a pre-defined threshold. The threshold for promotion to the AFC is the LFU count in the AFC. If the hit count in the annex cache exceeds the threshold, the flow is promoted to the AFC. The victim flow from the AFC is then placed in the annex cache. Finally on a miss

in the annex cache, a flow replaces the LFU flow of the annex cache.

When a counter saturates in the AFC, all the counters are shifted right by 1 place. This saturation control mechanism also serves as aging mechanism for flows so that the counts of unused flows are quickly reduced to make space for new flows. Although this aging mechanism performed well in experiments, a more robust agin mechanism like periodic right shifts could also be employed.

4.4.3 Overhead of the Aggressive Flow Detector

The overhead of the AFD mechanism is calculated using Cacti [89]. Table 4.1 shows the area, timing and power overhead of the AFD mechanism when a 512 entry annex cache is used. These results are calculated using 32 *nm* ITRS-HP device technology. This overhead is very small which makes the AFD mechanism very suitable to be used with the packet scheduler.

Access Time	Area	Power
0.375 ns	0.04 mm^2	0.0445 W

Table 4.1: Timing, area and power overhead of aggressive flow detector

4.4.4 Load Imbalance Detection

The length of the longest queue is used to detect the load imbalance in the system, i.e., when the length of the longest queue in the system reaches a predefined threshold, the load imbalance signal is asserted. As long as the load imbalance signal is asserted, all the aggressive flows from the overloaded

core are migrated to the least loaded core. The migrated flows are forwarded to the new core even after the load imbalance signal is de-asserted as a result of flow migration.

Most modern network processors have dedicated hardware units for management of packet queues [6, 67, 21] and a lot of research has been done on design of these hardware queue managers [92, 113, 76]. These queue managers implement different active queue management algorithms (e.g., Random Early Detection RED) and monitor the queue length as part of their normal operation. This queue length information can easily be used by the load balancer to detect the need for flow migration, i.e., it can easily be reported to the packet scheduler when the queue reaches a threshold. Hence, additional hardware resources are not needed to monitor queue length, because the queues are already monitored for congestion control purposes. In this work, it is assumed that the hardware queue manager monitors the queue state and generates the load imbalance signal.

4.5 Evaluation

4.5.1 Performance of Aggressive Flow Detector (AFD)

Accuracy of the AFD

Recall from Section 4.4.2 that the AFD has two components: An aggressive flow cache (AFC), and an annex cache. An annex cache can be viewed as a preliminary filter where non-aggressive flows are filtered out from entering the

small AFC. Therefore, any entry in the AFC is considered as an aggressive flow. The effectiveness of the AFD is evaluated by varying annex cache size while setting the size of the AFC constant at 16 entries. Since the AFC size is fixed, only the top 16 aggressive flows can be detected. A perfectly accurate AFC will hold the IDs of the top 16 aggressive flows. A flow found in the AFC that is not among the top 16 flows identified by off-line analysis is considered a false positive.

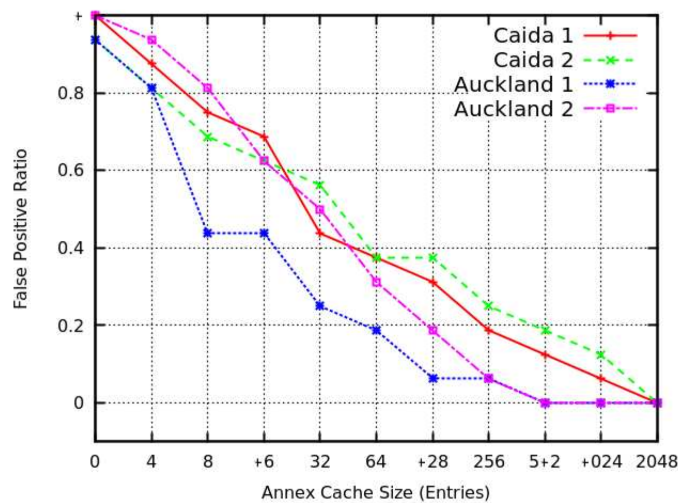


Figure 4.6: False Positive Ratio in a 16 entry AFC with varying annex cache size

Figure 4.6 shows the false positive ratio (false positives/total entries) in the AFC when the annex cache size is varied. The results are shown with a 4-way associative annex cache. Experiments were repeated with different configurations and a 4-way associative annex cache provides good performance. As the size increases, the annex cache can hold more flows to choose a possible candidate for promotion to the AFC. In other words, the pool of aggressive

flow candidates increases and the chances of aggressive flows residing in the cache for the AFC promotion becomes higher. For the Auckland traces, the AFC can identify all top 16 flows with 100% accuracy with a 512 entry annex cache. The Caida traces have more active flows and thus require a larger annex cache. In Caida 1 and 2 respectively, only 14 and 13 most aggressive flows are correctly identified with a 512 entry annex cache. When the size is doubled to 1024 entries, accuracy (false positive ratio) improved by an average of 6.25%. Although there are 2 or 3 false positives in Caida 1 and 2 cases, they are not random flows that are promoted to the AFC. In fact, when the 20 most aggressive flows are considered as an area of interest, these false positives fall into the aggressive flow category. Yet, for consistency of work, those flows are treated as false positives.

Effect of Window Size

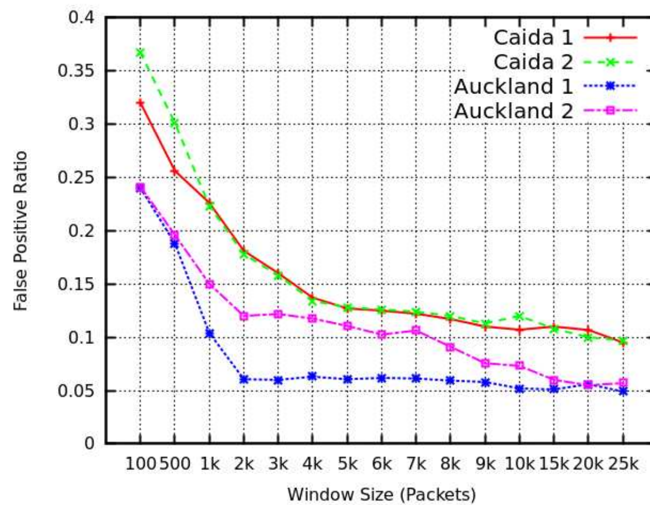


Figure 4.7: Effect of window size on accuracy of the AFD

Figure 4.6 looked at the accuracy of the AFD mechanism at the end of the simulations. Since LAPS needs to peek into the AFC whenever load balancing is required, another experiment is performed where the accuracy is checked at every fixed interval. In Figure 4.7, the same accuracy evaluation is performed with varying interval steps. In this experiment, the size of annex cache is fixed to 512 entries. The AFD shows accuracy above 90% from a small step size such as every 1000 packets to large step sizes. This implies that the AFC will contain the most aggressive active flows regardless of when it is accessed. In dynamic scheduling schemes like the proposed scheme, it is vital to maintain a high level of accuracy across the entire execution.

Effect of Sampling

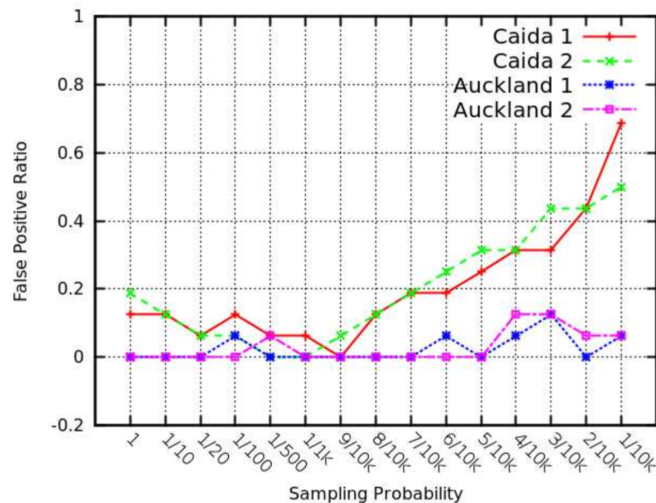


Figure 4.8: Effect of sampling on accuracy of Aggressive Flow Detector (AFD)

Figure 4.8 shows the false positive ratio when packets are sampled with

a probability p and not all of them access the AFD. It is interesting to note that false positive ratio improves initially with sampling. This is because sampling acts as a filter, i.e., the probability of large flows being sampled is higher than the smaller flows. However, the performance deteriorates for the Caida traces at larger sampling intervals. Sampling up to 1/1k probability gives better or equal performance than sampling all packets for all traces. The Caida traces generally have a large number of high data rate flows and hence their performance deteriorates if sampling is increased too much. Sampling not only improves the accuracy, but also reduces power consumption because now each packet does not have to access the AFD.

4.5.2 Benefit of Limiting Migration to Only Top Flows

In this section, the benefits of migrating only the most aggressive flows to achieve load balance are presented. The results are presented relative to Dittman's Load Balancer (DLB) which migrates arbitrary flows irrespective of their size in order to achieve load balance. To demonstrate the effectiveness of LAPS, simulations are conducted where only one service (IP forwarding) is active in the processor. Real network traces are used as input traffic to simulate the real flow scenarios. The input packet rate is set to slightly more than 100% of what this configuration can achieve under ideal conditions. Results are presented with for a traffic of period 60 seconds.

Figure 4.9 shows the number of packets dropped relative to DLB. A lot more packets are lost if no flow is migrated, but for almost all traces similar or

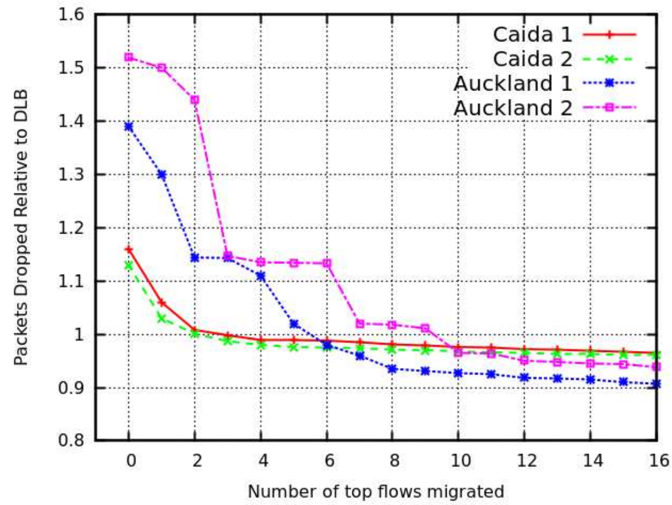


Figure 4.9: Packet drop relative to Dittman's Load Balancer (DLB) with varying the AFC size

better throughput than DLB can be achieved if only top 10 flows are identified and migrated. The real benefit of LAPS, however, is to maintain the order of the packets. Figure 4.10 shows that the percentage of out of order packets is reduced by 85% if only top 16 flows are identified and migrated. This benefit comes from minimizing the number of flow migrations as compared to DLB. In DLB, many non-aggressive flows are migrated that incur a flow migration penalty without providing any benefit in load balancing. In contrast, if only the most aggressive flows are migrated, load-balancing can be achieved by migrating only a small number of flows and thus out of order delivery of packets can be reduced. Figure 4.11 shows that the number of flow migrations are reduced by 80% if only the most aggressive flows are migrated. This reduction in flow migrations also means better D-Cache locality for both flow specific and routing data.

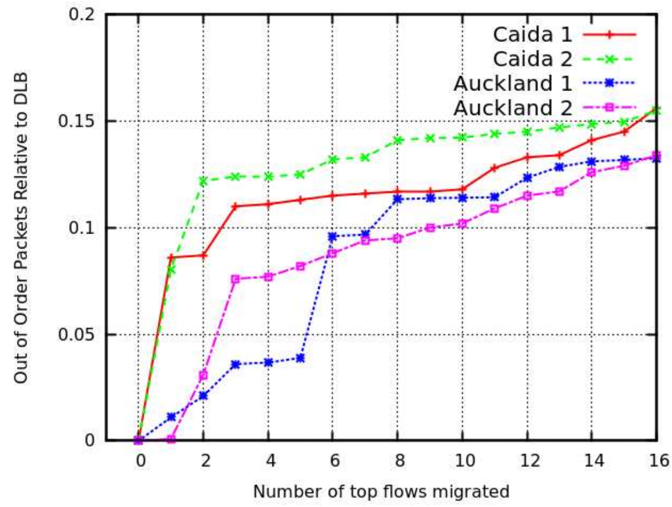


Figure 4.10: Number of out of order packets relative to DLB

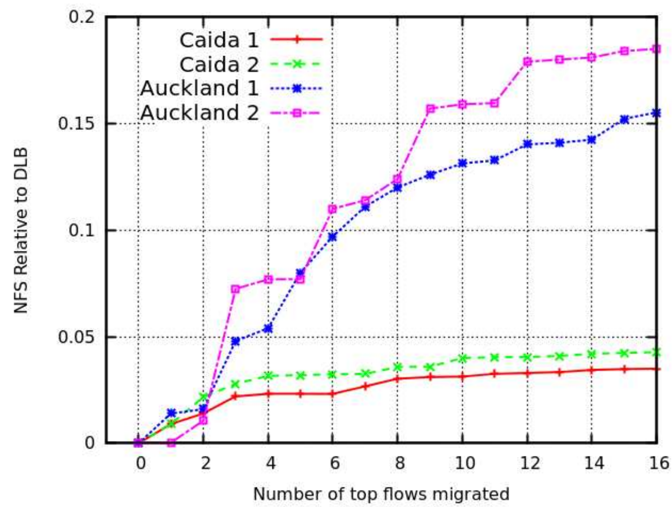


Figure 4.11: Number of flow shifts relative to DLB

4.5.3 Analysis of Flows on Migration

In order to understand the behavior of the system, the flows allocated to the overloaded core at the instance of flow migration are analyzed. Figure

4.12 shows the number of unique flows present in the queue of the overloaded core when a big flow is migrated from that core. Generally, a large number (15-20) of flows are present. This indicates that the overload is caused by a combination of large and small flows and migrating the large flow is expected to mitigate the load imbalance. A small number of flows would indicate that the overload is caused by small number of large flows and there is a potential that migrating the large flow would result in imbalance even in the newly allocated core. Figure 4.13 shows the number of big flows allocated to the overloaded core at the time of flow migration. From the plot it can be seen that the imbalance is usually caused by multiple big flows and migrating one flow is not expected to cause imbalance in the new core.

Presence of multiple big flows in the queue of overloaded core opens up the opportunity to further improve the scheme. For example, when migration decision is made preference could be given to the big flow which will cause less distortion in the order of packets, e.g., the flow with less number of packets in the queue could be preferred over the flow with larger number of packets. Such a scheme is likely to increase the complexity of the system becasue now core association of the flows and their number of packets in the system need to be monitored. Design of such a system is part of future work.

4.6 Summary

This chapter presents a design of a load balancer for the data plane packets of a network processor. The load balancer uses a hash table based design for

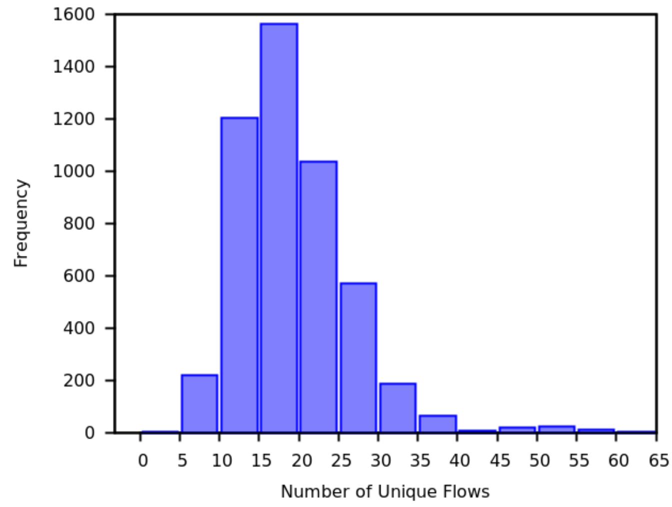


Figure 4.12: Number of flows allocated to overloaded core at the time of migration

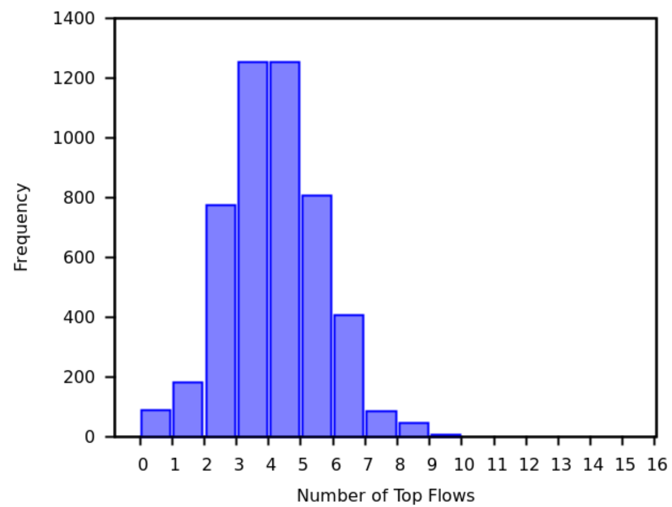


Figure 4.13: Number of big flows allocated to overloaded core at the time of migration

packet scheduling. It minimizes the number of flow migrations by limiting migration only to the top aggressive flows. The top aggressive flows are identified

using a novel two level caching scheme based on the annex cache. Results of experiments with real network traces are presented to show the effectiveness of the aggressive flow detector and the load balancer. In the next chapter, this hash based packet scheduler is extended to support multiple services in a router. The hashing scheme is also modified to support variable number of active cores.

Chapter 5

I-Cache Aware Packet Scheduling in Multi-service Routers

5.1 Introduction

The previous chapter presented design of a packet scheduler and load balancer for network processors. This chapter extends the design for multi-service routers. It begins with an introduction to the issues related to packet scheduling in multi-service routers in reference to instruction cache (i-cache) locality and dynamic allocation of cores to services. Design and evaluation of a packet scheduler is presented next. The next chapter presents power management techniques for network processors and presents the effectiveness of hash based

scheduler in the presence of power management techniques.

5.2 Scheduling Requirements in Multi-service Routers

Network processors are commonly employed in network edge packet processing systems. These edge routers are increasingly being required to support a rich set of services. These services include basic packet forwarding as well as more sophisticated services like intrusion detection, IPSEC encryption, IPSEC decryption, etc. An example workload for such a multi-service router is shown in Figure 3.2. The trend towards more functionality and complexity in data path processing is expected to continue [35]. Similar need for complex and varying functionality can be found in virtualized router platforms [39]. Virtual routers need to support several parallel networks with different data path functionality. This need for flexible processing with multiple services has led to deployment of network processors with highly parallel architectures and programmable cores.

A key challenge in these multi-service routers is packet scheduling. A packet scheduler needs to be aware of instruction cache locality in addition to flow locality. The data plane packet processing cores used in these processors are usually small with a small i-cache (8-16KB). These caches can only hold a single program at a time. The performance of a core will deteriorate due to i-cache misses if it has to process packets of different application types

[106]. The design of the packet scheduler presented in Chapter 4 is extended in this chapter to support multiple services with a goal to maintain the i-cache locality.

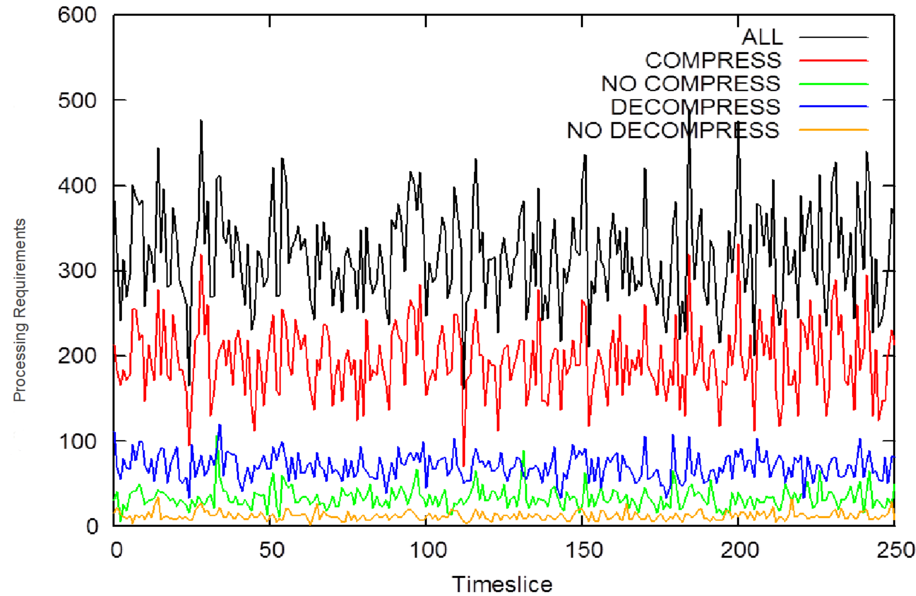


Figure 5.1: Variation in processing requirements in a multi-service router [65]

Another challenge for the hash based packet scheduler is dynamic adaptation of resources in multi-service routers. If cores are allocated to services statically at design time based on their worst case arrival rates, it will result in unnecessary hardware over-provisioning with high system cost. All services do not experience their worst case traffic at the same time, so most of the processing resources will be under-utilized [65, 107]. Figure 5.1 shows the variations in processing requirements of different services over time for a multi-service router [65]. A system that can multiplex cores among different services lowers

the total number of cores needed and reduces the system cost. In this chapter, the hash based scheme load balancer presented in Chapter 4 is extended to support dynamic adaptation of cores using incremental hashing.

Many researchers have presented dynamic adaptation techniques for multi-service routers [65, 107, 94, 97]. These schemes either require OS intervention or require expensive profiling information [107, 97, 108]. Data plane cores are generally devoid of OS and lightweight monitoring is required to handle the line rates. The resource manager presented in this chapter uses a queue based state monitoring framework similar to the one presented by Raghunath et al. [94]. The resource allocation scheme presented by Raghunath et al. uses a pipelined model for packet processing applications. In contrast, the resource allocator presented in this chapter considers packet processing as a single stage operation as described in Chapter 3. In addition, the dynamic resource allocator is integrated with the packet scheduler and a novel hashing scheme is presented to allow dynamic allocation and deallocation of cores to services over time.

5.3 Packet Scheduler for Multiservice Routers

This section presents the design of a packet scheduler that considers i-cache locality in addition to flow locality. This scheduler is called Locality Aware Packet Scheduler (LAPS). The design goals of LAPS are: a) to achieve high throughput by maintaining i-cache and flow locality, b) to minimize out of order departure of packets and c) to have a low overhead in order to sustain

high packet rates.

LAPS is an extension of the hash based load balancer and packet scheduler presented in Chapter 4. A simple hash based design as proposed in [33, 102] can result in inefficient i-cache usage. To avoid i-cache misses, LAPS partitions the map table among different services, i.e., each partition will have its own map table. All the cores in a single map table will always get packets that require the same processing so i-cache locality will be preserved.

In addition, LAPS also includes strategies to dynamically allocate different number of cores to services based on the traffic. The packet scheduler needs to utilize the newly allocated core while minimizing flow migrations. If all flows are blindly redistributed, it will result in a large number of flow migrations. Flow migrations are detrimental to performance and packet order as shown by experimental results in Chapter 4. LAPS introduces incremental hashing to manage the hash tables to deal with this situation which allows to dynamically allocate different number of cores to services while minimizing flow migrations. When the network traffic changes, the processing requirements of each service will vary. A dynamic adaptation system has two responsibilities: First, it should detect the changes in the processing requirements to trigger the adaptation. Second, it should make appropriate adaptations to handle those changes. The next few sections give details of how these two functions are performed by the proposed dynamic adaptation system.

5.3.1 Monitoring of Processing Requirements

In order to trigger dynamic adaptation, the system needs to collect information about processing requirements of individual service. This monitoring of processing requirements needs to have low overhead and should not affect the performance of packet processing applications. It also needs to react quickly to traffic variations in order to avoid packet loss.

For data plane processing, expensive performance counter based monitoring as used by some previous research [107, 108, 97] is not feasible. In contrast, packet arrival and departure rates from the queues are monitored. As explained in Chapter 4, such monitoring can be embedded in the hardware queue and buffer managers, which are present in almost all modern network processors. In order to measure the arrival rate (R_a) for a service, an arrival counter ($count_a$) is incremented whenever a packet is added to the queue. After a fixed interval of t seconds, the rate can be calculated as $R_a = count_a/t$. Similarly, rate of departure can be calculated as $R_d = count_d/t$.

Per service queues may not be implemented separately, but input queues to individual cores can be used to measure the arrival and departure rates of each service. The resource manager has complete system knowledge and it can aggregate the stats of multiple cores allocated to a service to get per service arrival and departure rates. Such a resource manager can be implemented as a software on one of the data plane cores and it can poll the queue mangers periodically to get the per queue statistics. The arrival and departure rates of queues gives a direct indication whether more resources are needed for a

service or not.

5.3.2 Core Allocation Policy

At initialization, cores are equally divided among services. As traffic varies over time, the requirements of each service change and the core allocation needs to be modified. This situation arises when packet arrival rate for a service is greater than the departure rate, i.e., $R_a > R_d$. Furthermore, the processing ability of a single core (R_{d1}) in packets per second can be obtained through profiling information. The required number of cores, C, for this service are calculated as

$$C = R_a / R_{d1} \tag{5.1}$$

The value of C is scaled to the nearest integer value. If C is greater than the current allocation of cores, K, this service is marked as a needy service with demand of $C - K$ additional cores. Notice that if R_a is slightly higher than R_d, C will be floored to the value of K and no request is made for additional cores. The queues will start to fill up because R_a is greater than R_d. The resource manager waits until the input buffer occupancy reaches some $high_{th}$ to allocate the additional core. This helps to avoid unnecessary allocations due to bursty traffic. The resource manager of LAPS needs to find an additional core to fulfill the demands of a requesting service and update the map table accordingly. There could be two possible situations when requests for additional cores are made:

Underload Condition

The underload condition occurs when current demand for processing resources is less than the total available processing resource. Under this situation, some services will have more resources than their demands. The resource manager keeps a list of services that have indicated more resources than their demands (Section 5.3.3). The service that indicated surplus resources for the longest period of time is chosen as the victim service. This policy makes sure that the deallocated core has the least utility for the victim service.

Overload Condition

The overload situation happens when current demand for processing resources is more than the available resources. When the system is overloaded, cores are allocated to services in proportion to their demands. Specifically, a service is considered eligible for more cores only if current allocation is less than its proportional share. If current requirement for core for a service i is C_i and current number of cores allocated is N_i, service is eligible for additional cores only if $K_i/N < C_i/\sum C_i$. Otherwise, any additional demands are ignored.

5.3.3 Core Release Policy

Underload Condition

When the value of C in Equation 5.1 goes below the current allocated cores K, a timer starts. When the timer reaches $Idle_{th}$, one core is marked surplus

by adding it to a list of extra cores. $Idle_{th}$ helps to avoid de-allocation due to a transient dip in the traffic. The core still remains allocated to the same service. If the same service needs more resources in near future, this core can be unmarked and removed from the list of surplus cores without incurring the overhead of a context switch. The value of $idle_{th}$ is set to 10μs based on the previous research [23].

Overload Condition

In this condition, a service can be victimized even if its current allocation is not more than the requirements but it holds more cores than its proportional share, i.e., $K_i/N > C_i/\sum C_i$. This policy is adapted from the policy presented in [94]. The policy is adapted to work in the run-to-completion model and novel scheme for load redistribution on allocation and deallocation of cores is presented.

5.3.4 Load Redistribution on Core Allocation

When an additional core is allocated to a service, the resource manager appends the core ID to the end of the hash table for that service, i.e., the hash table size grows by 1. To minimize the number of flows migrated on core allocation, LAPS makes use of *Linear Hashing* (also known as Incremental Hashing). This scheme allows a hash table to grow one bucket at a time and does not require rehashing of all flows currently allocated. This makes it useful for load balancing because it is desirable to minimize the flow disruption

when an additional core is allocated to a service. The Linear Hashing scheme was introduced by [74] and has been described in [54]. Following is a brief introduction of how this scheme works.

Initial Assignment of Flows

The linear hashing scheme has m initial buckets labelled 0 through $m-1$, and an initial hashing function $h_0(k) = f(k)\ mod\ m$ that is used to map any key k to one of the m buckets, and a pointer p that points to the bucket to be split whenever new bucket is added. Initial value of p is 0. An example is shown in Figure 5.2. The example follows the same pattern as the example given in linear hashing tutorial by Zhang et al. [112]. In this example, $h_0(k) = k\ mod\ m$ is used as a hash function for simplicity, where $k\ mod\ m$ indicates the least non negative integer remainder of k/m.

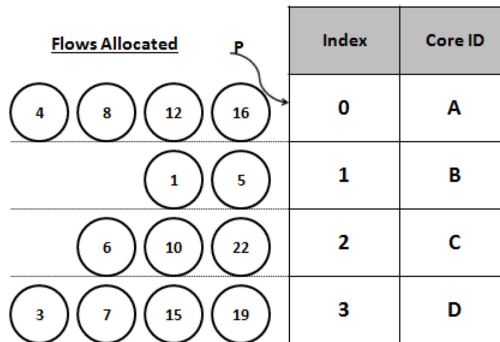


Figure 5.2: Initial assignment of flows. $m = 4$, $p = 0$, $h_0(k) = k\ mod\ 4$.

Bucket Split

When the first additional core is added to the hash table, bucket 0, that is pointed by p, is split into two buckets: the original bucket 0 and a new bucket m. The flows originally mapped to bucket 0 by hash function h_0 are now distributed between bucket 0 and m using a new hash function h_1. Figure 5.3, shows layout of linear hashing after the new core bucket has been added to the map table. The shaded flows are the flows that are moved to the new bucket. Bucket 0 has been split and and the flows originally in bucket 0 are distributed between bucket 0 and bucket 4, using a new hash function $h_1(k) = k\ mod\ 8$.

When another additional core is allocated, i.e., another bucket m+1 is added to the hash table, the flows mapped to bucket 1 will now be redistributed using $h1$ between buckets 1 and m+1. A crucial property of $h1$ is that the keys that were mapped to some bucket j by h_0, are remapped to either j or bucket $j + m$. This is a necessary property for linear hashing to work. An example of such hashing function is: $h_1(k) = k\ mod\ 2m$.

Round and Hash Function Advancement

After enough core allocations, all original m buckets will be split. This marks the end of splitting round 0. During round 0, p went from 0 to $m - 1$. At the end of round 0, there are $2m$ buckets in the hash table. Hash function h_0 is no longer needed because all $2m$ buckets can be addressed by h_1. Variable p is reset to 0, and a new round, namely round 1, starts. A new hash function h_2 needs to be used. Figure 5.4 shows the state of hash table at the end of

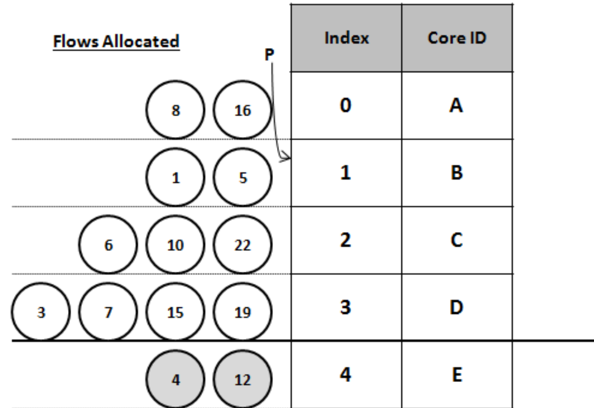


Figure 5.3: Flow redistribution after allocation of an additional core. $p = 1$, $h_0(k) = k \; mod \; 4$, $h_1(k) = k \; mod \; 8$.

splitting round 0.

In general, the linear hashing scheme uses a family of hash functions h_0, h_1, h_2, and so on. Let the initial function be $h_0(k) = f(k) \; mod \; m$, then any later hash function is $h_i(k) = f(k) \; mod \; 2^i m$. This way it is guaranteed that if h_i hashes a key to the bucket $j \in [0..2^i m - 1]$, h_{i+1} will hash the same key to either j or bucket $j + 2^i m$. At any time, two hash functions h_i and h_{i+1} are used. In general, in splitting round i, hash functions h_i and h_{i+1} are used. At the beginning of round i, $p = 0$ and there are $2^i m$ buckets. When all those buckets are split, splitting round $i + 1$ starts, p goes back to zero, the number of buckets become $2^{i+1} m$, and hash functions h_{i+1} and h_{i+2} will start to be used.

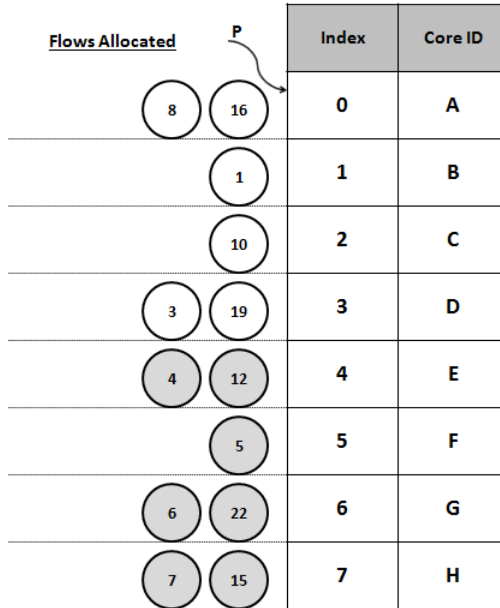


Figure 5.4: Flow redistribution at the end of round 0 (beginning of round1). $p = 0$, $h_1(k) = k \ mod \ 8$, $h_2(k) = k \ mod \ 16$

Summary and Mapping Scheme

Initially, each service is allocated m cores, i.e., there are m buckets in the hash table. At any time the hash table manager has the following components:

1. A variable i that indicates the current splitting round.

2. A variable p that points to the bucket to be split next.

3. A total number of $2^i m + p$ buckets in the hash table.

4. Two hash functions h_i and h_{i+1}. The base hash function used is CRC16, i.e., $f(k) = CRC16(k)$.

Whenever a packet arrives, the hash scheduler has to map it to one of the buckets in the map table. The mapping scheme works as follows:

$$h(k) = \begin{cases} h_{i+1}(k) & : h_i(k) < p \\ h_i(k) & : h_i(k) \geq p \end{cases}$$

i.e., if $h_i(k) \geq p$, choose bucket $h_i(k)$ because this bucket has not been split yet in the current round. If $h_i(k) < p$, choose bucket $h_{i+1}(k)$. The value of p is incremented whenever a new core is allocated to the service. Use of this incremental hashing in conjunction with load balancing scheme of Section 4.4.1 allows us to add additional cores to a service with minimal disruption to the existing flows.

5.3.5 Load Redistribution on Core Release

When a core is reallocated to another service, it is removed from the bucket list of the victim service. Essentially, a process that is a reverse of the load redistribution on allocation takes place. The value of round i is updated, i.e $i = (b/m) - 1$, where b is the current number of buckets in the map table. The value of p is set to $b - 2^i m$. and the hash function is also changed accordingly.

5.3.6 Overall Scheme

Figure 5.5 shows the overall architecture for LAPS. The bucket list in the mapping table for each service S_i is dynamic and the dynamic size b_i changes with traffic variations. The hash function for each service is decided based on

the size of its bucket list. Following steps are taken when a packet arrives:

1. If the flow ID hits in the migration table, the packet is forwarded to the core ID indicated by the migration table.

2. If the flow ID does not hit the migration table, the map table is searched using the hash function and the packet is forwarded to the core indicated by the mapping table.

3. Under a load imbalance, the aggressive flows (flows that hit in AFC) are migrated to the least loaded core allocated to that service similar to the load balancing scheme of Chapter 4.

4. When the number of cores allocated to a service is insufficient, the bucket lists are updated. An idle core is removed from the bucket list of the donor service and is added to the bucket list of overloaded service.

5.3.7 Timing Analysis of LAPS

To sustain a traffic of 100 Gbps, the scheduler has to be able to schedule 100 million packets per second (considering mixed sized packets). Note that the critical path of LAPS is Hash Delay $\rightarrow$ Map Table Access $\rightarrow$ Mux Delay. The AFD and the map table update are not part of the critical path since they work in the background. The critical path is dominated by the hash delay. Researchers have shown that CRC16 can be calculated very efficiently and can easily operate at the speed of 200 MHz [33, 4]. The delay of map table

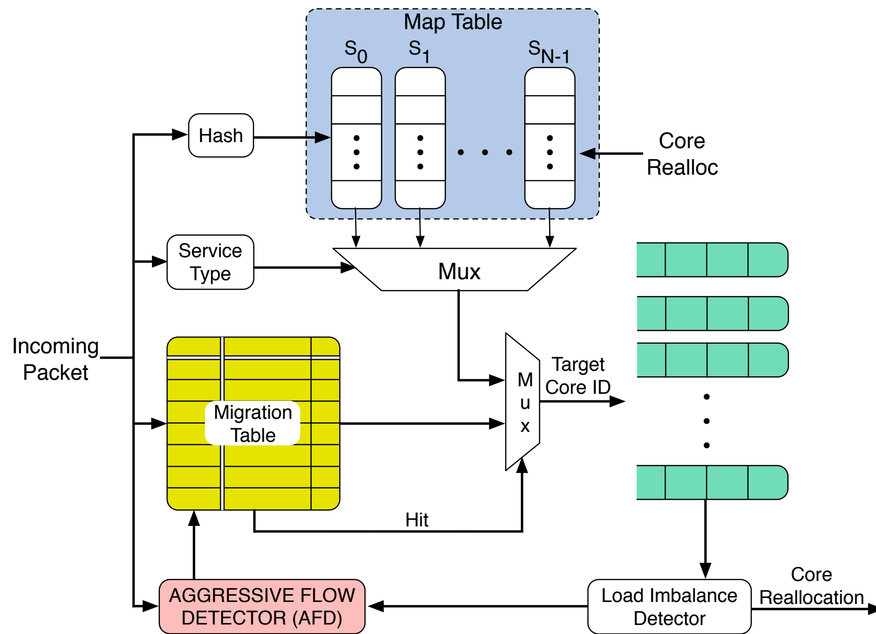


Figure 5.5: Locality Aware Packet Scheduler

access is a fraction of a nano second according to Cacti [89] simulations. This means LAPS is capable of sustaining at least 200 million packets per second and hence is capable of handling future traffic of even beyond 100 Gbps. For even higher traffic rates in the future, multiple copies of hash function could be used. A straight forward implementation is to use a separate hash function for each application. The ability of hash based schedulers to sustain such high data rates makes them very suitable for growing internet traffic rates.

5.4 Evaluation

5.4.1 Throughput Improvement with LAPS

LAPS aims to improve throughput by exploiting locality in instruction and data caches. Figure 5.6 shows effectiveness of LAPS in improving throughput of a sixteen core system. In this experiment, all four services of Figure 3.2 are active. The simulation infrastructure of Figure 3.1 is used. The traffic rate generator is configured to increase the traffic gradually to measure the maximum throughput supported by the system. Traffic is equally divided among the four services, i.e., Path 1 through 4 of Figure 3.2. Caida 1, Caida 2, Caida 3 and Caida 4 traces are used for generating packets for Path 1, Path 2, Path 3 and Path 4. Figure 5.6 compares throughput of LAPS with a First Come First Served Sc(FCFS) and an Arbitrary Flow Shift (AFS) scheduler (See section 2.3.1 for description of AFS scheduler). The X-axis shows the combined input traffic rate which is equally divided among all the services and y-axis shows the traffic rate observed at the output.

The FCFS scheduler services packets in their arrival order and does not consider flow or instruction locality. As a result, it causes many data and instruction cache misses and results in the worst throughput among the three schedulers. As compared to FCFS, AFS reduces some flow migrations and is able to improve throughput a little, but AFS is still unaware of instruction locality and results in suboptimal performance. In comparison to these two schemes, LAPS improves both flow and instruction locality and results in sub-

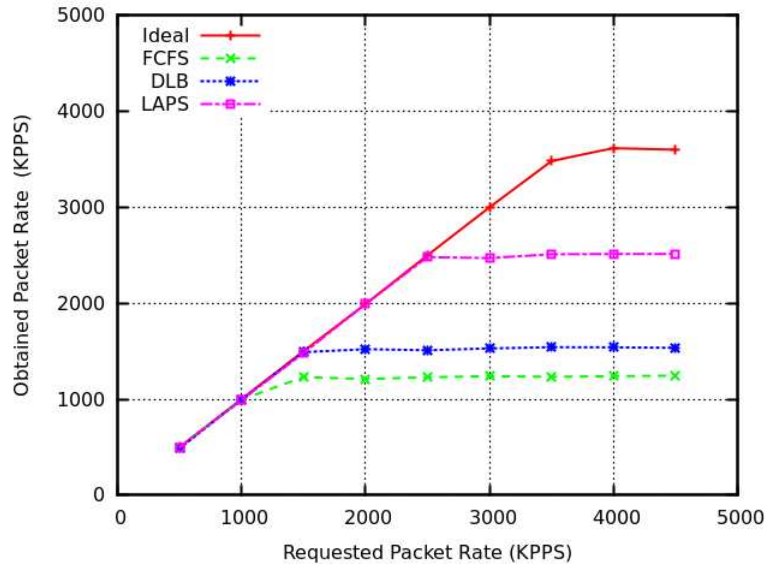


Figure 5.6: Throughput comparison of different schedulers

stantially better throughput (56% more than AFS and about 100% more than FCFS). Ideal throughput represents a system with no cache miss penalties. The plot is obtained by setting the cache miss penalties to zero. Although such a system is infeasible, it represents a theoretical maximum which can be achieved if the system has full knowledge of everything and is able to move data and instructions into caches before they are needed.

Note that the throughput supported by the simulated sixteen core system is much less than that of the industrial system. There are two reasons for this: First, the software implementations of services are taken from open source benchmark suites whereas companies use highly optimized implementations. Second, the experiments are based on software only implementations and do not use hardware accelerations.

5.4.2 Overall Performance Improvements with LAPS

Figure 5.6, presents results with increasing traffic. In this section, the experiment is repeated where the traffic rate is governed by equation 3.8. The results with two sets of parameters for equation 3.8 are presented. These parameters are listed in Table 5.1. Set 1 represents the underload condition, i.e., the aggregate traffic rate is less than the ideal capacity of 16 cores. Set 2 represents an overload condition, i.e., the data rate is more than the capacity of the 16 core system.

	Service	a	b	C	m	σ
Set 1	S1	0.6	0.03	0.3	40	0.1
	S2	0.7	-.025	0.1	25	0.05
	S3	0.3	0.01	0.07	60	0.25
	S4	0.1	0.005	0.09	600	0.3
Set 2	S1	1.2	0.002	0.3	100	0.3
	S2	1.0	-.02	0.15	25	0.05
	S3	07	0.004	0.25	30	0.25
	S4	0.4	0.01	0.18	200	0.3

Table 5.1: Parameters governing traffic rate. Rate is in Mpps and period is in seconds

For each service, real network traces listed in Table 5.2 are used to generate the input packets. The combination of sets of parameters in Table 5.1 and traces in Table 5.2 creates different traffic scenarios listed in Table 5.3. Figure 5.7 shows packets dropped with three schemes under the traffic scenarios shown in Table 5.3 for a traffic of 60 seconds. LAPS outperforms FCFS and AFS in both the under-load and overload conditions due to its superior instruction and data cache locality. Figure 5.8 compares the i-cache

Group	S1	S2	S3	S4
G1	Caida1	Caida2	Caida3	Caida4
G2	Caida5	Caida6	Caida2	Caida3
G3	auck1	auck2	auck3	auck4
G4	auck5	auck6	auck7	auck8

Table 5.2: Traces used in experiment for packets of individual services

Scenario	Parameter Set	Trace Group
T1	Set 1	G1
T2	Set 1	G2
T3	Set 1	G3
T4	Set 1	G4
T5	Set 2	G1
T6	Set 2	G2
T7	Set 2	G3
T8	Set 2	G4

Table 5.3: Different traffic scenarios used in experiments

performance of these schemes.

FCFS and AFS distribute packets of different services arbitrarily to cores and suffer from poor i-cache locality. Figure 5.8 shows percentage of packets that experience cold caches, i.e., the percentage of packets that require different processing than the previous packet processed by the same core. FCFS and AFS schemes drop packets even in under-load conditions because almost 60% of packets suffer from cold cache penalties. On the other hand, LAPS partitions the cores among services effectively and enjoys good i-cache performance. Under overload scenarios (T5 through T8), LAPS also suffers from some cold caches because cores are dynamically switched between

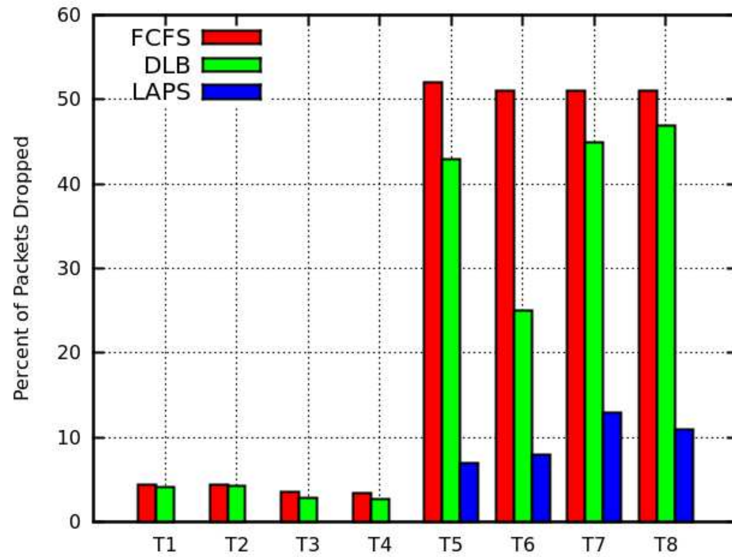


Figure 5.7: Comparison of packet dropped under different traffic scenarios

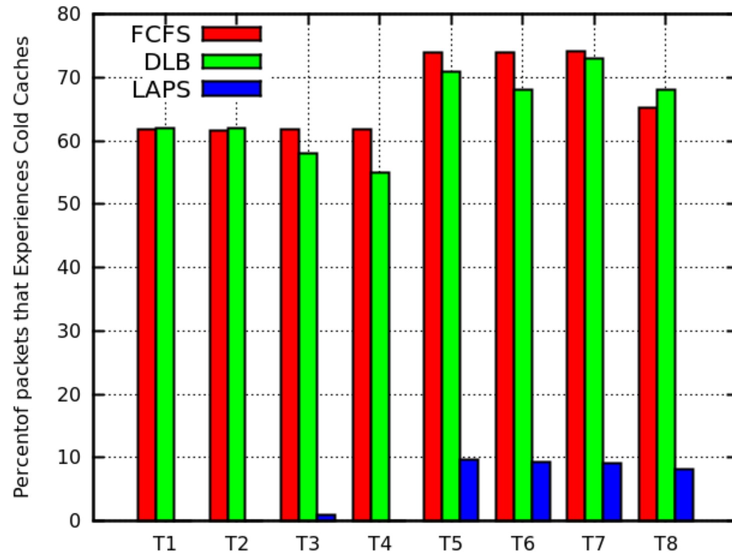


Figure 5.8: Comparison of i-cache performance under different traffic scenarios

services based on traffic variations.

Apart from the throughput improvement, another major advantage of

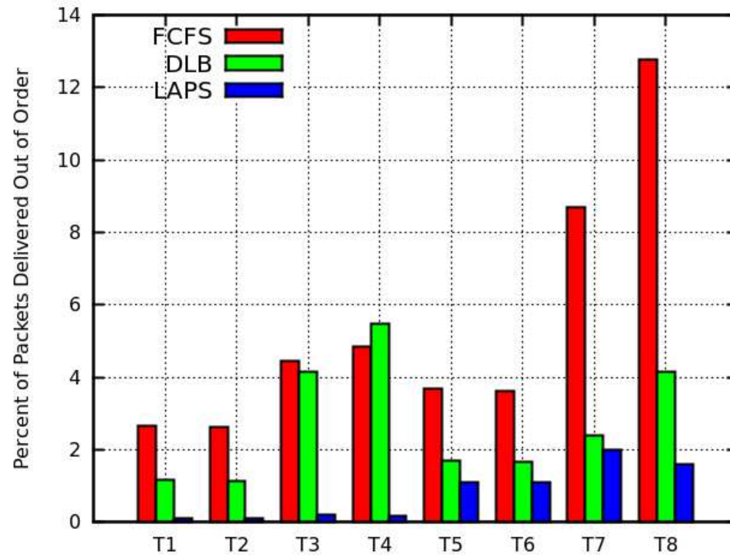


Figure 5.9: Comparison of packet order under different traffic scenarios

LAPS over the other two schemes is its capability to maintain packet order. Figure 5.9 shows the percentage of packets that are delivered out of order. LAPS effectively maintains packet order by minimizing the flow migrations and maintaining the flow locality.

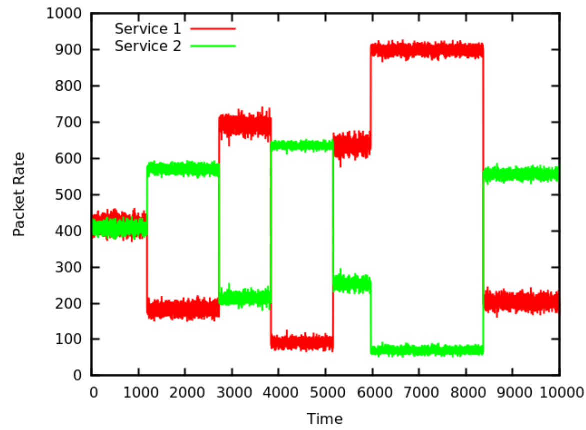
5.5 Dynamic Behavior of the System

In order to observe the effectiveness of of dynamic resource allocation scheme, temporal behavior of number of cores allocated to each service is plotted. Figure 5.10 shows the dynamic behavior when two services are active in the system. Service 1 is the same as Path 1 of Figure 3.2, i.e., the outgoing VPN traffic is encrypted using IPSEC encryption. Service 2 is the Path 3 of Figure 3.2 which corresponds to processing incoming packets through a firewall. The

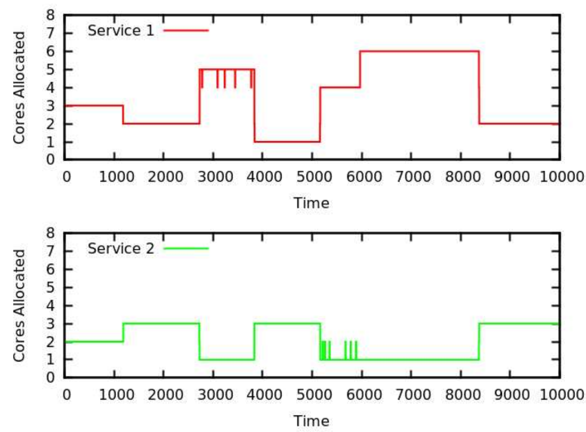
traffic requirements of each service are varied over time and the response of the resource allocation system is observed. Figure 5.10 shows that the system is very effective in following the changing traffic requirements and changes the core allocations to match the demands of each service very effectively.

5.6 Summary

This chapter extends the design of packet scheduler presented in Chapter 4 for multi-service routers. A dynamic resource allocation scheme is presented that allocates cores to services based on dynamic traffic requirements. The packet scheduler is extended to support this dynamic allocation of cores while minimizing flow migrations. The next chapter deals with techniques to exploit dynamic variations in traffic to save power.



(a) Traffic variation over time



(b) Number of cores allocated over time

Figure 5.10: Temporal behavior of the resource allocator

Chapter 6

Efficient Traffic Aware Power Management

6.1 Introduction

The previous two chapters have focussed on the performance improvement of network processors. This chapter explores how to exploit variability in traffic the volume to save power in the network processors. Some basics of power management were presented in Chapter 2. In this chapter, a Traffic Aware Power management scheme (TAP) is presented. Use of traffic prediction for the purpose of predictive power management is also explored. Furthermore, this chapter also presents the benefits of careful packet scheduling and resource allocation in terms of power savings.

6.2 Motivation

6.2.1 The Power Management Problem

The rise in traffic demands and complexity of applications have resulted in an increase in the complexity and number of cores in network processors. As a result, power consumption is becoming a major problem and the energy consumption in routers is reaching the limits of air cooling [22, 19]. For example, a fully configured Cisco CRS-1 core router can consume up to one megawatt of power [19]. A typical router has a set of line cards and each line card has one or more network processors [11, 3, 81]. Power consumption of a single line card can reach up to 500 Watts [3, 9]. Modern routers can have hundreds of line cards. For example, the CISCO CRS-1 router can house up to 1152 line cards in different chassis. These line cards are densely packed in routers. High power consumption can result in high temperature of parts and failure due to thermal stress. Such failures affect the reliability and availability of networks. This results in lower quality of service and increased expenditures in replacement parts. High power consumption of equipment leads to higher cooling costs and results in increased operational expenditure of the network. With increasing traffic rate demands and computational complexity, the number and complexity of cores in network processors is on the rise, resulting in more and more power consumption. Tight power budgets and dense integration requirements call for the design of power efficient network processors.

6.2.2 Power Saving Opportunities

The multicore packet processing systems are usually designed and provisioned with enough resources to satisfy peak traffic load, but network traffic varies with time and reaches the peak value for only a small portion of time. Studies have shown that network utilization is under 30% even for backbone networks [7]. Figure 6.1 shows traffic observed over two days by the CAIDA monitor [30] at the internet backbone in Chicago. There is a huge variation in packet rates and thus different processing requirements at different times of the day. Most of the time, the traffic rate is below the maximum traffic and it is not necessary to run the processors at their full capabilities. The low activity periods can be exploited to save power in network processors by running them in low power modes and/or by turning off some processing cores.

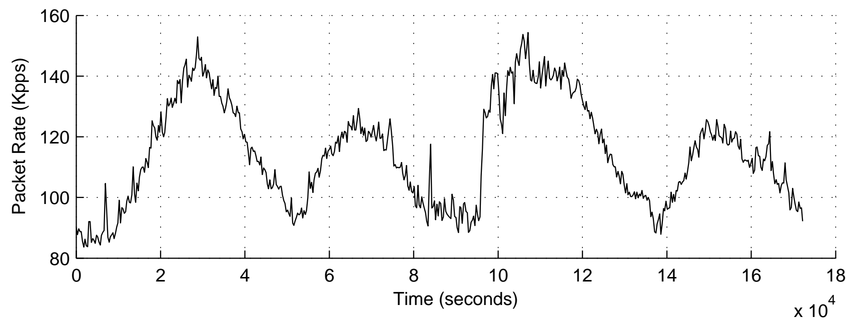


Figure 6.1: Variation in traffic arrival rates(Kilo Packets Per Sec) over 2 days at equinix-chicago internet backbone

6.2.3 Ineffectiveness of Existing Schemes

Power management in network processors is in its infancy. Different efforts are being made to apply power management schemes used in general purpose processors to network processor power management. These schemes, however are not designed for network processor workloads and result in un-optimal power managements. These existing power management schemes use some proxies to estimate the performance requirements, e.g., CPU utilization or idle time. These schemes are either too conservative, resulting in loss of power saving opportunities or are simply unaware of traffic demands and result in extra packet loss. The proposed power management scheme, TAP, uses direct information of traffic for power management. TAP also employs traffic prediction to manage power predictively. TAP uses a traffic and a load predictor to pro-actively change the number of active cores. A predictive power management scheme can provide more power efficiency than a reactive scheme because power adaptations can be applied before the load changes. Traffic prediction, however, is non trivial since network traffic can vary at different time scales, e.g., short term variations and time of the day variations. Secondly, mispredictions can result either in loss of power saving opportunities (over-prediction) or packet loss (under-prediction). TAP needs a mechanism to deal with such mispredictions and variation of traffic within the prediction interval. Also, TAP needs to make power management decisions within a fraction of power adaptation interval. This means that traffic predictor and power management algorithms cannot be too complex. In order to overcome these challenges, TAP uses a

low cost traffic and load predictor. TAP also uses Dynamic Voltage and Frequency Scaling (DVFS) to adjust the frequency of the active cores to adapt to the variation of traffic during the prediction interval.

6.3 Traffic Aware Power Management (TAP)

An efficient power management scheme for network processor has to make the following decisions:

1. Predicting the load for the next interval
2. Deciding the required number of active cores N_{opt} for the predicted load
3. Deciding the frequency f_i for each active core i.

This section provides details of the proposed TAP scheme and explains how TAP makes the three above mentioned decisions.

6.3.1 Prediction of Computing Requirements

The computational requirements for network processors depend on both traffic arrival rate and the computation complexity of the applications. TAP uses a new parameter called *Traffic_factor* that combines both traffic rate and computational complexity to give a true estimation of load to be handled by the network processor.

Traffic Prediction

TAP uses Double Exponential Smoothing Predictor (DES) for traffic prediction. A more complex predictor may result in greater accuracy in some situations but a low overhead predictor is desirable for energy efficiency. The Double Exponential Smoothing (DES) predictor is a low overhead predictor with accuracy comparable to complex predictors like Artificial Neural Network (ANN) or Wavelet transform based predictors. This makes DES very suitable for this application. This study uses the DES predictor, but designers can use the proposed scheme with any other predictor based on their requirements. This section gives a brief introduction to the DES predictor and a short comparison of different predictors is presented in Section 6.4.3.

Double Exponential Smoothing (DES) Predictor

Exponential Smoothing assigns exponentially lower weights to older observations. Single exponential smoothing does not work well when there is a trend in the data [8]. A trend means that the average value of the time series increases or decreases with time. However, Double Exponential Smoothing adds a trend component for estimation and is considered more appropriate for data with trends. The equation for DES based prediction for a time series $X(t)$ is given as

$$X_{t+1} = S_t + b_t \tag{6.1}$$

where

$$S_t = \alpha X_t + (1 - \alpha)(S_{t-1} + b_{t-1}) \tag{6.2}$$

and

$$b_t = \gamma(S_t - S_{t-1}) + (1 - \gamma)b_{t-1} \tag{6.3}$$

S_t and b_t are the smoothed value of the stationary process and the trend value respectively. S_t and b_t are added together to get the prediction for next interval. α defines the speed at which older values of S_t are damped. When α is close to 1, dampening is quick and when α is close to 0, dampening is slow. γ is similar smoothing constant for b_t. The values of α and γ are obtained using non-linear optimization techniques and are learned during the training phase of the predictor. Note that this is a very low cost predictor. It requires only four registers for storing α, γ, S_{t-1} and b_{t-1}. To make a prediction, it requires six multiplications and four addition operations. This low overhead and reasonable accuracy make it an appropriate predictor for the purpose of power management.

Traffic Factor (β)

TAP uses a new parameter called *Traffic_Factor* that combines the traffic rate and the application's processing capability to give a true estimation of the processing requirement. The traffic rate is the rate at which packets arrive at the input and is represented as Packets Per Second (PPS). It is naturally tempting to use traffic or packet rate directly to exploit traffic variability, but different applications have different processing requirements and hence can support different packet rates, i.e., a complex application will require more resources to sustain a particular traffic rate than a simple application. This means packet

rate cannot be used directly for power management purposes. If processing requirements are known in addition to the packet rate, this information can be used to drive the power management scheme. If Cycles Per Instruction (CPI) and Instructions Per Packet (IPP) are known, Cycles Per Packet (CPP) required can be directly calculated, i.e.,

$$CPP_{req} = IPP \times CPI \tag{6.4}$$

Traffic_Factor (β) is defined as

$$\beta = \frac{PPS_{predicted} \times CPP_{req}}{(max_cpu_freq \times total_cores)} \tag{6.5}$$

where $PPS_{predicted}$ is the traffic predicted using the DES predictor explained above. The Traffic Factor, β, incorporates both the application performance requirements and the traffic rate into a single parameter and is an excellent parameter for use in the power management schemes.

Note that the CPP is independent of the frequency level, i.e., the cycles per packet will remain constant with changing frequency and hence this parameter does not suffer from the same limitations as the processor utilization used by previous schemes. Processor utilization is a direct function of frequency whereas CPP does not depend upon frequency under the assumption that there are limited off-chip memory accesses. This assumption is valid in network processors since the packet processing applications are small and fit into caches and the packet queues are also implemented via on-chip memories

[17]. Furthermore, no additional resources are needed to measure this parameter. Many network processors like P4080 provide performance counters to measure PPS, IPP and CPI directly [10].

6.3.2 Deciding Number of Active Cores

The required number of active cores C can be directly calculated from the Traffic Factor (β) as

$$C = \beta \times total_cores \tag{6.6}$$

The number of active cores are adjusted as shown in Listing 6.1. The sampling interval used is 500 μS.

Listing 6.1: Algorithm to decide number of active cores

```
1 for (every sampling interval){
2    C = β * total_cores;
3    if (C > active_cores )
4       wakeupCores(C - active_cores);
5    else if (C < active_cores)
6       killCores(active_cores - C);
7 }
```

If C is less than the current number of active cores, all extra cores can be shut down. When a core goes into a sleep state, it first goes into C1, where it stays for 2 sampling intervals and then it goes to state C2. TAP

uses C2 as the deepest sleep state. Note that the difference between the ladder governor and TAP is that they use different input parameters. Instead of using idle time, TAP uses the *Traffic_Factor* to drive the C-state management. In order to wakeup cores, TAP looks at the input queue length in addition to the *Traffic_Factor* (See Listing 6.2). If the size of the queue length reaches a certain threshold, TAP wakes up one of the sleeping cores.

6.3.3 Deciding Frequencies of Active Cores

If the load predictor over-predicts, or if there is a variation in the traffic during the prediction interval, DVFS can be used to further save power at smaller timescales. At regular intervals (50 μs), TAP checks the size of the input queue and based on the size of the queue, decides whether to increase or decrease the power levels. When a packet arrives at the input interface, the interface control logic stores the packet in the input queue until it is picked and serviced by an available processor.

If the input queue is nearly empty most of the time, enough resources are available to handle the traffic load. If it is near full, it means that more processing capability is needed. The length of the input queue gives a direct indication of whether more resources are needed or not. The algorithm for finding the appropriate P-state is shown in Listing 6.2.

If the queue is nearly empty, it is assumed that the system has excess processing capability and the frequency is reduced to the next lower level. When the queue starts to grow, it means that the current processing capability

is lower than what is needed and frequency level is increased.

Listing 6.2: Algorithm to find P-state values

```
1  int pstate[numcores];
2  for (every sampling interval){
3    if (avg < low_th){
4      core = findMin(pstate);
5      pstate[core] = pstate[core]+1;
6    }
7    else if (avg < high_th){}
8    else{
9      core = findMax(pstate);
10     if(core == -1)
11        WakeUpCore(1);
12     else
13        pstate[core] = pstate[core]-1;
14   }
15 }
```

Note that Listing 6.2 uses a global governor which makes decisions based on the queue size instead of having a separate governor for each core. The array $pstate[numcores]$ holds the P-states of each core. Each core can have five possible P-states similar to Table 2.1. The functions $findMin()$ and $findMax()$ return the indices of the fastest and the slowest cores, respectively. If the queue size is less than the threshold value low_{th}, TAP lowers the frequency of

the fastest core and if it is higher than the $high_{th}$, TAP increases the frequency of the slowest core. Also note that if the frequency cannot be increased further, TAP increases the number of cores. This allows adjustment to dynamic variation in traffic during the interval or under prediction and helps to avoid dropping any packets.

6.3.4 Measuring Queue Length

TAP uses the average queue length during the sampling interval to make decisions about choosing the appropriate P-states. A low pass filter is used to calculate the average queue size as proposed in the RED algorithm [40]. Thus a short term increase in traffic, which results from bursty traffic or transient network congestion, does not affect the average queue length. The filter used is an exponential smoothing filter and is given as

$$avg = \alpha \times qlength + (1 - \alpha) \times avg \tag{6.7}$$

where $qlength$ is the instantaneous size of the queue and α defines the speed at which older values are dampened. An α of 0.025 is used in this study.

Many congestion control algorithms like RED [40], rely on the occupancy of the input queue and modern network processors provide hardware support for these congestion control algorithms. The P4080 processor provides dedicated hardware (QMan) for queue management. QMan implements a variant of the RED algorithm and keeps track of the input queue length.

Thus queue length can be used for power management purposes and it does not require additional resources since it is already monitored for congestion control purposes. Even if it is not readily available in any processor, it is easy to add functionality in the interface logic for this purpose.

6.3.5 Deciding Threshold Values

TAP monitors the length of the input queue and compares it with predefined threshold values to decide the state of the processor. These threshold values are allowed to adjust dynamically according to the changing traffic load and thus it is not needed to find a common value for all traffic rates and applications.

The optimum value of $high_{th}$ depends on the wake up time for a core in deep sleep state. When $qlength$ reaches this threshold value and all the active processors are running at highest frequency, additional cores need to be turned on in order to avoid dropping packets. Assuming the wakeup time of 200 μs, extra buffer space is needed which is enough such that no additional packets are dropped before an additional core is on line. The maximum observed packet rate in this work was about 200 KPPS. Assuming this is the worst case increase in the packet rate, an additional buffer space for 40 packets is needed to store the packets before the core is on line. If the maximum queue size is Q_{max}, $Q_{max} - 40$ is used as the value of $high_{th}$ in these experiments. The value of Q_{max} is set to 80.

The value of low_{th} can be chosen from a wide range. Essentially, it should not be too close to $high_{th}$ to prevent $avg_qlength$ from oscillate between

low and high thresholds. low_{th} is chosen to be such that $high_{th} = 4 \times low_{th}$. Essentially, P-state manager tries to keep the input queue less than 50% full all the time, but if the traffic rate is higher than what the P-states can manage, the queue length will increase beyond $high_{th}$. When the queue length exceeds this value, it means that the current number of processors is not enough to manage the traffic and an additional processor is woken up. If the value of queue length reaches 95%, it means that the new processor was turned on too late and the values of C_{th}, low_{th} and $high_{th}$ are decreased by 10%. If *qlength* never reaches C_{th} value for 10 consecutive intervals, all the thresholds are increased by 10%.

6.4 Evaluation

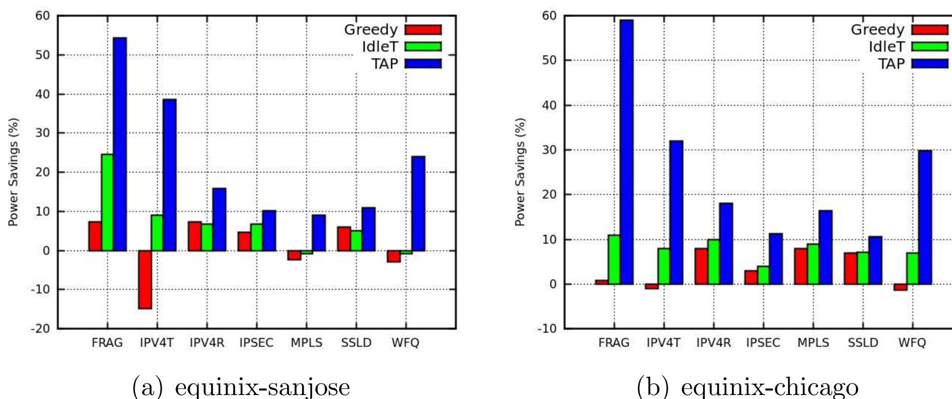
6.4.1 Power Savings with Real Network Traces

In this study, different power management policies are compared. Table 6.1 lists the policies under consideration. The C-state policy in the Base scheme is similar to the Linux ladder governor. The thresholds used are based on Table 2.2. The P-state management policy is based on the Linux ondemand governor explained in Listing 2.1. The Greedy scheme uses a similar ladder governor for C-state management, but the P-state manager is a greedy algorithm of Section 2.4.3 that tries to minimize EPI.

The IdleT policy uses a scheme similar to one proposed by Luo et al. [81, 82] for managing the number of active cores based on number of idle threads in

Policy	C-state Policy	P-state Policy
Base	ladder	ondemand
Greedy	ladder	greedy EPI
IdleT	idle threads	ondemand
TAP	Traffic Factor based	Queue Length based

Table 6.1: Power management policies implemented for comparison



(a) equinix-sanjose (b) equinix-chicago

Figure 6.2: Power savings on real network traces.

the given interval. For a fair comparison with TAP, this scheme is modified to go into a deeper sleep state if it remains in the current state for two consecutive intervals. Note that the original scheme just made use of clock gating and did not utilize deeper sleep states. This scheme is further augmented to use the ondemand governor for frequency management of individual cores. TAP is the proposed scheme based on traffic prediction.

Figure 6.2(a) and Figure 6.2(b) show the power savings for the equinix-sanjose and equinix-chicago traces, respectively. The results are presented relative to the Base policy. Table 6.2 shows the absolute numbers for power consumption during the trace. It can be seen that TAP consistently beats the

other policies on all applications. A major portion of the energy saving comes from having the right number of active cores. Table 6.3 shows average number of active cores throughout the trace for the different strategies. TAP scheme has the lowest average number of active cores on all benchmarks. *Traffic Factor* allows TAP to estimate the minimum number of active cores as compared to other schemes that base their decisions on idle time. TAP changes the number of cores proactively and thus reduces the lag between load changes and power adaptation. Other schemes, being reactive in nature, lose some power saving opportunities since the cores are turned off after some idle time has been elapsed. The Base and tje Greedy schemes result in the highest number of active cores. the C-state management in ithe Base and the Greedy schemes is too conservative in the sense that it waits for break even time to elapse before going to the deep sleep state. Also note that the number of active cores depends on the P-state management as well. For example, if the active cores are running at a lower speed than needed, they will result in activating more cores than required. TAP calculates required number of active cores directly using the traffic factor and thus results in the minimum number of cores being active. For the IdleT scheme also, the number of active cores is a function of frequency of individual cores and results in higher number of cores than needed. In some situations, Greedy results in more cores than any other policy. The reason is that P-state policy in this scheme is totally unaware of traffic. It tries to minimize the energy per instruction irrespective of the traffic rate. Thus it results in more cores since individual cores are running

at lower frequency. Consider the situation presented in Figure 6.3. The input packet rate is such that a single core is required to operate at power level P1 to sustain that traffic.

	equinix-sanjose				sanjose-chicago			
	Base	Greedy	IdleT	TAP	Base	Greedy	IdleT	TAP
FRAG	7.82	7.29	5.93	3.53	7.4	7.32	6.6	3.8
IPV4T	6.73	7.79	6.11	4.10	6.73	6.91	6.23	4.5
IPV4R	29.80	27.80	28.1	25.23	31.5	29.1	28.3	26.1
IPSEC	63.20	60.69	59.80	56.59	71.5	69.3	68.7	62.4
MPLS	45.03	46.6	45.2	40.8	55.5	52.2	51.1	46.3
SSLD	101.6	95.4	96.1	90.1	99.4	94.1	96.1	90.1
WFQ	15.94	16.73	16.02	12.1	15.8	16.3	14.7	11.1

Table 6.2: Power consumption in Watts for different schemes

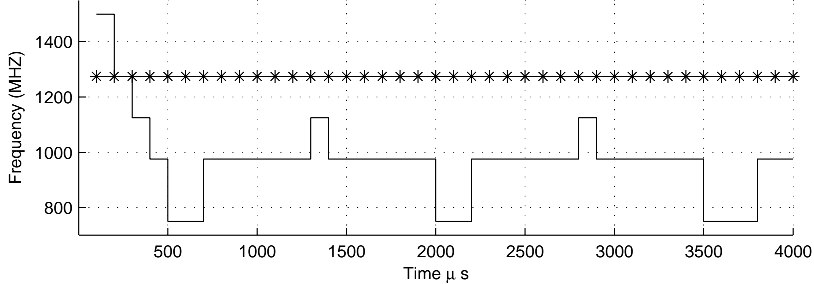


Figure 6.3: Greedy algorithm to minimize EPI. Asterisks show optimum power level

Figure 6.3 shows the response of greedy algorithm described in Section 2.4.3 to this input traffic. Since this greedy algorithm is unaware of the traffic requirement, it continues to move in the direction of lowering the EPI and overshoots the required power level and operates at frequency lower than the required frequency for the rest of the trace. This scheme will result in running

2 cores instead of 1 for the above situation, which will lead to more power consumption.

Application	Base	Greedy	IdleT	TAP
Frag	3.10	2.81	2.21	1.30
IPV4T	3.34	3.69	3.10	1.42
IPV4R	7.70	7.90	7.62	5.50
IPSEC	29.10	30.10	29.00	24.30
MPLS	13.81	15.72	13.81	10.31
SSLD	30.53	31.21	31.11	26.11

Table 6.3: Average number of active cores

6.4.2 Effectiveness of DVFS

Although most of the benefit comes from having the right number of active cores, the ability of individual cores to change frequency also provides some power benefits. Figure 6.4 presents a comparison of proposed TAP scheme with and without the capability of DVFS. In the scheme without DVFS, the number of cores are controlled by the traffic aware scheme and there is no DVFS, i.e., all the active cores run at full speed. Although most of the benefit comes from having the right number of active cores, DVFS still has a significant impact on performance. From the figure it can be seen that, in most cases, DVFS improves the power consumption by 15% and as much as by 39% in case of FRAG. This benefit comes from the fact that if TAP over predicted the workload or there is variation in traffic during the prediction interval, then DVFS helps to reduce the power by lowering the frequencies of the cores. Figure 6.5 shows potential benefit of having the ability to change frequency

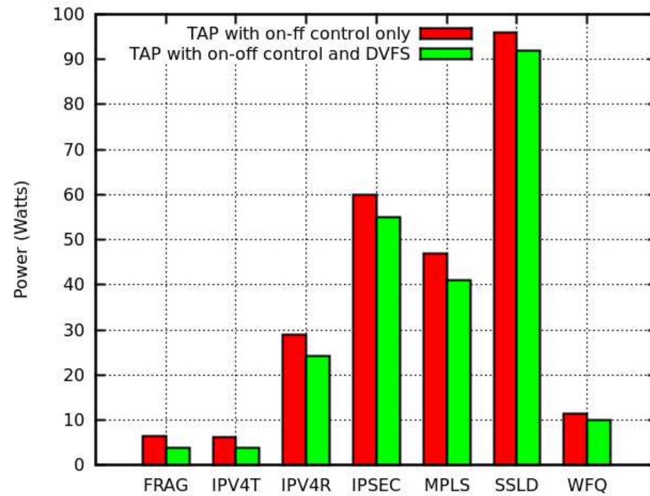


Figure 6.4: Benefit of combining DVFS with on-off control

and voltage of the cores in addition to adjusting the number of active cores. The plot is for a 16 core system running IPV4R benchmark when traffic is varied from 0 to 100% that the given configuration can handle. The power consumption is plotted relative to the system that has ability to change number of active cores, but does not have a per core DVFS. It is observed that there is a lot of potential power saving at low to medium traffic. At high traffic, obviously there is less room for power savings. The dotted line shows potential power savings if there is global DVFS, i.e., all cores change their frequency in unison and at any moment all cores are at the same frequency. Global DVFS provides a good tradeoff between design complexity and power savings since it decreases the design complexity and is able to exploit most of the power saving opportunities. The P-state algorithm becomes even simpler if global DVFS is used instead of per core DVFS, i.e., instead of using the *findMin()* and

findMax() functions, the frequencies of all cores can be increased or decreased together based on the queue length.

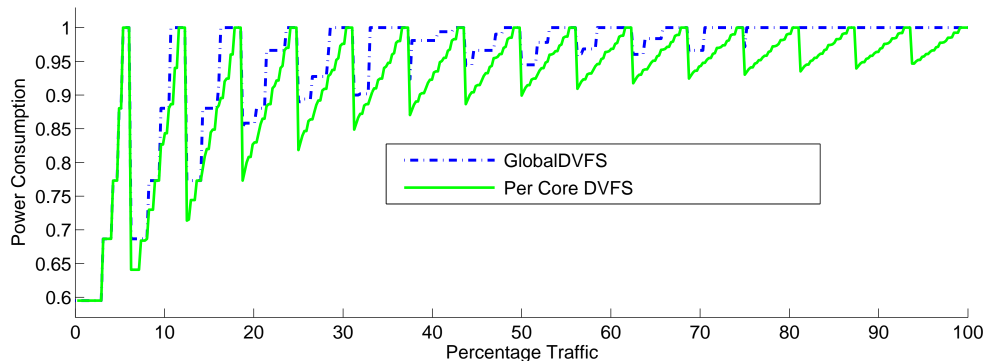


Figure 6.5: Global DVFS vs per core DVFS

6.4.3 Effectiveness of DES Prediction

A distinguishing feature of TAP is that the traffic_factor is based on the DES predictor for traffic (Section 6.3.1). Figure 6.6 compares the accuracy of different predictors. The predictors under comparison are listed in Table 6.4.

Predictor	Description
LV	Last observed value is used as prediction for next interval
MA	Moving average of last 8 observations
AR	Auto Regression based prediction
ARMA	AutoRegressive Moving Average of order
ANN	Artificial Neural Network (3 layers)
DES	Double Exponential Smoothing

Table 6.4: Traffic predictors compared

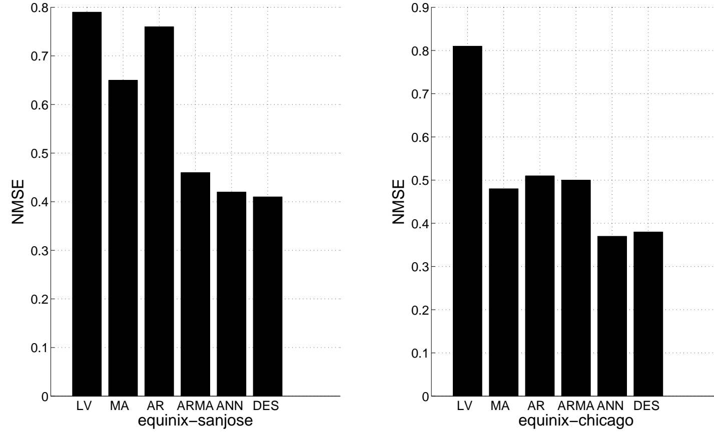


Figure 6.6: Accuracy comparison of traffic predictors. Graphs show NMSE values (lower bar is better).

The Normalized Mean Square Error ($NMSE$) is used to compare the performance of predictors. This metric is widely used for evaluating prediction performance. It is the ratio of mean square error to the variance of the series.

$$NMSE = \frac{1}{\sigma^2} \frac{1}{M} \sum_{t=1}^{M} (X_t - \hat{X}_t)^2 \tag{6.8}$$

where X_t is the actual value of traffic during interval t, $\hat{X}_t$ is the predicted value of X_t and M is the total number of predictions. σ^2 is the variance of time series during prediction. This metric compares the performance of the predictor with a trivial predictor (one that always predicts the mean of the time series). For this trivial predictor (mean predictor) $NMSE = 1$. If $NMSE > 1$, this means that the predictor is worse than the trivial one. $NMSE = 0$ in case of a perfect predictor.

From Figure 6.6, it can be see that DES performs comparably well as

compared to more complex ANN based predictor and it outperforms other predictors in the study by big margin. More details about different traffic prediction techniques can be found in [59]. Figure 6.7 shows the power savings

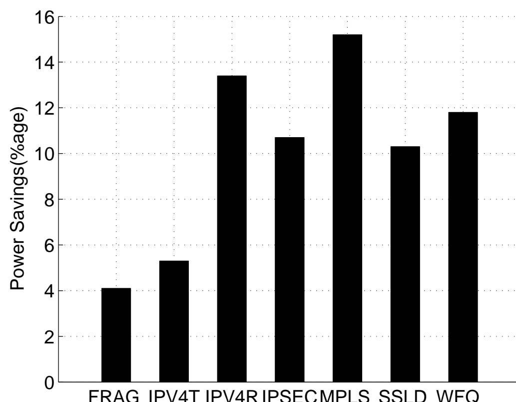


Figure 6.7: Power saving when using DES Predictor compared to LV predictor. The trace used is equinix-sanjose.

with DES predictor over LV predictor. The figure shows that a good predictor can result in more efficient management of power.

6.4.4 Packet Queue Behavior

Figure 6.8 shows the queue length values during a portion of the equinix-sanjose trace. It can be seen that the filtered queue length effectively neglects the short term variation in traffic and is effective in preventing the system from oscillating between states. Also, the scheme is able to adapt with increasing traffic, i.e., if the queue length increases above the threshold values, the system is able to adapt its resources to brings the queue length back within desired limits.

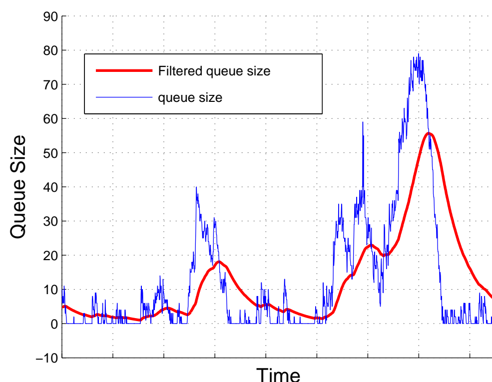


Figure 6.8: Queue size and filtered queue size

6.4.5 Power Saving at Different Traffic Rates

Figure 6.9 shows comparison of different power management schemes at different traffic rates with synthetic traces. The proposed scheme adapts well at different traffic rates and is the best performer for all applications at all rates. It is important to note that TAP does not gain much benefit from prediction in these traces. These traces are of constant data rate. Reactive schemes, which behave similar to LV predictor, are also able to accurately predict traffic. For expensive applications like SSLD and IPSEC, there is not much room for power saving since the load is already high, but TAP is able to get a benefit of around 6-10% even for those applications. In general, the Greedy scheme runs individual cores at lower power levels, but results in activating more cores. For IPSEC and MPLS Greedy scheme seems to perform similar to TAP at some data rates. The power numbers are similar, but the Greedy scheme results in 11% packet loss while TAP scheme does not drop any packets.

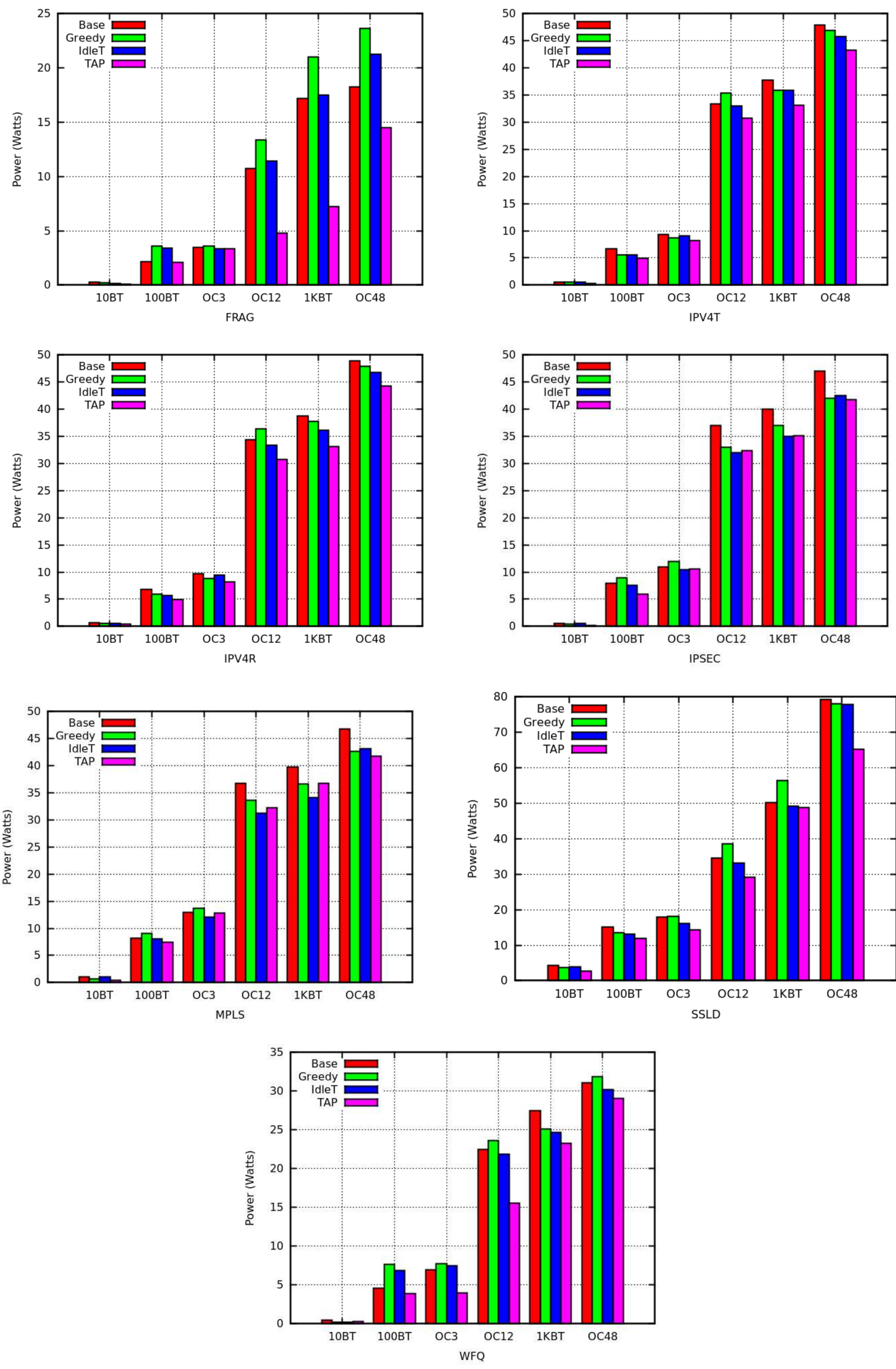


Figure 6.9: Power consumption comparison of different power management techniques at different traffic rates

6.4.6 Effect of Packet Scheduling on Power Savings

TAP has so far used First Come First Served (FCFS) scheduling of packets from a single global queue. Whenever a core becomes idle, it gets the packet from the head of the queue for processing. FCFS scheduling can result in poor performance as shown in Chapters 4 and 5. Careful packet scheduling, such as LAPS, can result in better per core performance. An experiment was conducted to observe the effect of LAPS on power consumption of the system. Figure 6.10 shows the overall scheme. Note that the LAPS scheme is targeted for the data plane cores which usually do not have the capability of DVFS. So this experiment is conducted with the assumption that the cores have on-off control only. Furthermore, these cores are generally devoid of an operating system, hence, the traffic factor (β) cannot be calculated. In this experiment, the resource manager of Chapter 5 is responsible for putting additional cores to deep sleep states based on the core release policy discussed in the previous chapter.

Figure 6.11 shows the effect of careful packet scheduling on power consumption on a 16 core processor. Figure 6.11 compares power consumption of FCFS scheduling with LAPS when the application running is IPV4T. Input traffic rate is governed by Equation 3.8 with base line set to 6 MPPS and other parameters are same as Set1 and service S1 of Table 5.1. Figure 6.11 shows that more power can be saved when careful packet scheduling is done. This is because per core performance improves when packets are scheduled carefully to exploit data and instruction cache locality. Hence the same work can be

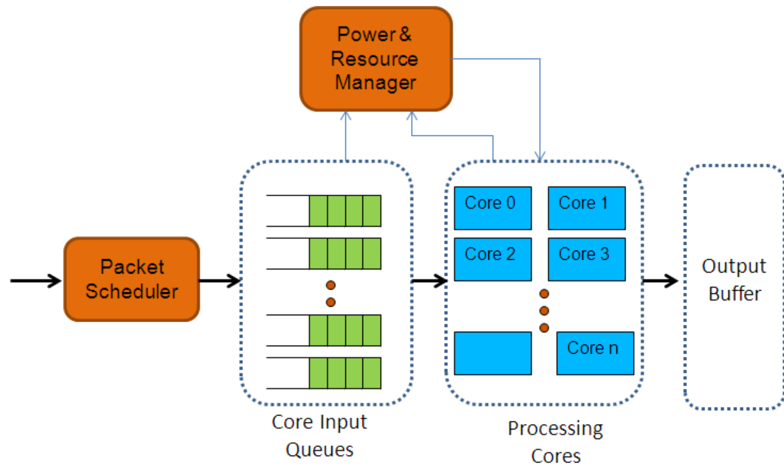


Figure 6.10: Combining LAPS and TAP

done by less number of cores and rest of the cores can be turned off to save power.

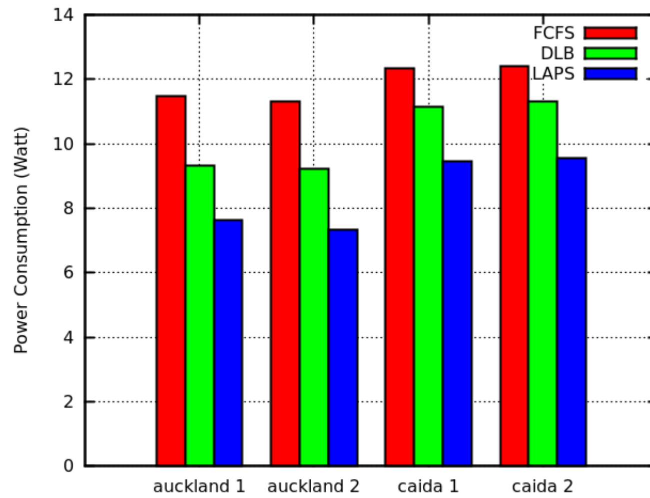


Figure 6.11: Power saving benefits of careful packet scheduling

Figure 6.11 considers only a single service active on the router. Figure 6.12 shows power consumption comparison when multiple services are active.

This figure shows power consumption with the traffic scenarios of Table 5.3. Significant power savings are observed at traffic scenarios T1 through T4 because LAPS is able to handle the traffic with less number of cores as compared to FCFS. For rest of the traffic scenarios, no power gains are observed. Although LAPS results in better performance in terms of packet loss (See chapter 5), all the cores are active and no power savings are observed.If we consider per packet power, LAPS results in 30% less power consumption per packet on average as compared to the DLB scheme for traffic scenarios T5 through T8.

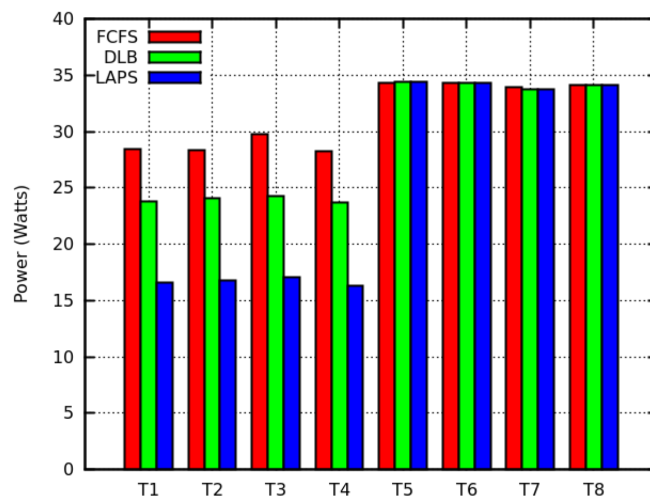


Figure 6.12: Power saving benefits of careful packet scheduling on a multiservice router

6.5 Summary

This chapter has presented a traffic aware power management technique for network processors. The technique changes both the number and frequency

of the active cores in order to save power during low traffic times. The next chapter concludes this dissertation and provides some areas for future work.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The study was set out to explore the techniques to efficiently utilize the large number of parallel cores in network processors. Network processors, due to their high performance and programmability, have become the main computing elements in networking applications like enterprise, core and edge routers. Routers are the main building blocks of networking. These routers have to support high traffic rates and multiple complex services. Performance of the network processors is critical for routers in order to deal with these demands. This study sought to answer two main issues related to performance optimization of network processors: (1) How to allocate work to multiple cores to optimize performance? (2) How to meet the desired performance requirements power efficiently? The study focussed on dynamic adaptations based on run time traffic behavior in order to optimize the performance. The study ad-

vances the state-of-the-art in the field of network processor design by making three contributions.

First, the design of a hash based packet scheduler and load balancer is presented in order to achieve the goals of preserving flow locality and packet order. A hash based packet scheduler performs very well in order to achieve these goals because it schedules the packets at the flow level and thereby inherently maintains packet order and flow locality. A serious impediment to performance of the hash based scheduler is the presence of skewed flow sizes in network traffic. Such skewed distribution of flow sizes can result in overloading some cores and may result in packet loss. To avoid packet loss, a load balancer is designed that migrates some flows from the overloaded cores to under-utilized cores. Flow migrations are undesirable because they result in bad data locality and can result in out of order packets. The load balancer proposed in this study minimizes the number of flow migrations by restricting migrations only to the aggressive flows.This strategy helps to achieve load balancing with minimum flow disruptions.

Second, the design of a dynamic resource manager is presented that allocates processing cores to multiple services based on dynamic traffic variations. This dynamic resource management scheme improves performance efficiency in two ways. First, partitioning the cores among services helps to improve performance by exploiting I-Cache locality. Second, multiplexing cores among multiple services reduces the overall core provisioning level. The resource manager is also integrated with the packet scheduler to optimize the per core

performance. Furthermore, the use of incremental hashing is proposed which is a low cost way of adding and removing cores to services while minimizing flow migrations.

Finally, the design of a power management scheme is presented that exploits variability in traffic volume to save power. The Power manager can work seamlessly with the resource manager because incremental hashing allows putting additional cores into sleep modes with minimum overhead. Also, integration of the packet scheduler results in even more power savings because the number of required cores to handle the input traffic is reduced if the performance of the individual cores is optimized by exploiting instruction and data locality.

The schemes presented in this dissertation show promising improvements over the previous work. Hash based designs of the packet scheduler and the resource manager have very low overhead. This makes the designs very scalable for data rates of 100 Gbps and beyond.

7.2 Future Research Directions

The proposals made in this dissertation can be adapted or extended for further improvement in network processor performance. Some potential areas of future research are outlined below.

7.2.1 Thermal Hot Spot Reduction

Since the power management scheme in this dissertation proposes to power up and power down cores dynamically, some cores will be used more than other cores. Such a behavior can be studied in the future work and power management scheme could also be made thermal aware so that no individual core becomes too hot. A tradeoff study needs to be done since one goal is to maintain locality and migrating work from a hot core to a cold core may incur intolerable overheads.

7.2.2 Multiple Scheduling Decisions per Packet

The processing of a packet may involve multiple steps. For example a packet may spend a few initial cycles on the cores, the next few cycles on a hardware accelerator and then some more cycles back on the core. This dissertation has assumed that a packet does not relinquish a core until its processing is finished even if the processing involves some offloading to a hardware accelerator. However in order to better utilize the resource, the core can start processing another packet if the previous packet is offloaded to a hardware accelerator for processing. In that case multiple scheduling decisions have to be made per packet: on entry into the system and upon every completion of core or accelerator processing. Extending the scheduler to make these multiple scheduling decisions can be studied in future.

7.2.3 Fairness and Quality of Service

The packet scheduler presented in this dissertation tries to minimize packet loss while minimizing packet reordering. In the future, this scheduler can be integrated with a QoS mechanism like the Resource Reservation Protocol (RSVP), that dictates to each router how to handle each packet (flow) based on its reservation or Differentiated Services (Diffserv) which treats different classes of packets differently.

Bibliography

[1] AMD Opteron Processor Power and Thermal Datasheet. http://support.amd.com/us/Processor_TechDocs/30417.pdf.

[2] Broadcom 64 core processor iBCM-88030. http://www.broadcom.com/press/release.php?id=s666869.

[3] Cisco 7609-s router. http://www.cisco.com.

[4] Cyclic redundancy code generator. http://www.actel.com.

[5] Energy Efficient Platforms - Considerations for Applications Software and Services. http://download.intel.com/technology/pdf/Green_Hill_Software.pdf.

[6] The Freescale P4240 processor. http://www.freescale.com.

[7] IPMON sprint. the applied research group. http://ipmon.sprint.com.

[8] NIST/SEMATECH e-handbook of statistical methods. http://www.itl.nist.gov/div898/handbook/.

[9] Power consumption for MX960. http://www.juniper.net.

[10] PowerQuiccIII Monitors. http://www.freescale.com.

[11] Quidway NetEngine 5000e. http://www.huawei.com.

[12] Red hat enterprise linux 6 power management guide. http://docs.redhat.com.

[13] Tilera 72 core network processor tile-gx72. http://www.tilera.com.

[14] The University of Auckland traces. http://wand.net.nz/wits/auck/2/.

[15] Using the linux cpufreq subsystem for energy management. http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaai/cpufreq/liaai-cpufreq_pdf.pdf.

[16] XLP832 multicore processor. http://www.broadcom.com.

[17] Intel IXP hardware reference manual, January 2003.

[18] Global action plan report. http://www.globalactionplan.org.uk, 2006.

[19] System power challenges. http://www.slidefinder.net/c/cisco_routing_research/seminar_august_29/1562106, 2006.

[20] Processor power management in windows vista and windows server 2008. http://www.microsoft.com, November 2007.

[21] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J. F. Logan, M. Peyravian, M. A. Rinaldi, R. K. Sabhikhi, M. S. Siegel, and M. Waldvogel. IBM PowerNP network processor: Hardware, software, and applications. In *IBM Journal of Research and Development*, volume 47, pages 177 –193, March 2003.

[22] J. Baliga, K. Hinton, and R. Tucker. Energy consumption of the internet. In *Joint International Conference on Optical Internet, 2007 and the 2007 32nd Australian Conference on Optical Fiber Technology, COIN-ACOFT 2007*, pages 1 –3, June 2007.

[23] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,*, volume 8, pages 299 –316, June 2000.

[24] W. Bircher and L. John. Core-level activity prediction for multicore power management. In *IEEE Journal on Emerging and Selected Topics in Circuits and Systems,*, volume 1, pages 218–227, 2011.

[25] W. L. Bircher and L. K. John. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 327–338, New York, NY, USA, 2008. ACM.

[26] J. D. Brutlag. Aberrant behavior detection in time series for network

monitoring. In *Proceedings of the 14th USENIX conference on System administration*, LISA '00, pages 139–146, Berkeley, CA, USA, 2000. USENIX Association.

[27] Z. Cao, Z. Wang, and E. Zegura. Performance of hashing-based schemes for internet load balancing. In *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM 2000*, volume 1, pages 332–341 vol.1.

[28] L. Chiaraviglio, M. Mellia, and F. Neri. Reducing power consumption in backbone networks. In *IEEE International Conference on Communications, ICC 2009*, pages 1–6, 2009.

[29] G. Chuvpilo, D. Wentzlaff, and S. Amarasinghe. Gigabit ip routing on raw. In *Proceedings of the 8th Intl. Symp. on High-Performance Computer Architecture, Workshop on Network Processors*, 2002.

[30] K. Claffy, D. Andersen, and P. Hick. The CAIDA anonymized 2011 internet traces. `http://www.caida.org/data/passive/passive_2011_dataset.xml`.

[31] P. Crowley and J. L. Baer. A modeling framework for network processor systems. In *Proceedings of the 8th Intl. Symp. on High-Performance Computer Architecture, Workshop on Network Processors*, 2002.

[32] G. Dhiman and T. S. Rosing. Dynamic power management using machine learning. In *Proceedings of the 2006 IEEE/ACM International*

Conference on Computer-Aided Dsign, ICCAD '06, pages 747–754, New York, NY, USA, 2006. ACM.

[33] G. Dittmann and A. Kerkersdorf. Network processor load balancing for high speed links. In *Proceeding of Intl. Symp. on Performance Evaluation of Computer and Telecommunication Systems*, 2002.

[34] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski. System-on-chip environment: a specc-based framework for heterogeneous mpsoc design. In *EURASIP J. Embedded Syst.*, volume 2008, pages 5:1–5:13, New York, NY, United States, January 2008. Hindawi Publishing Corp.

[35] W. Eatherton. Push of network processing to the top of the pyramid. In *Keynote Presentation at ACM/IEEE Symposium on Architecture of Network and Communications Systems (ANCS)*, 2005.

[36] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. In *ACM Trans. Comput. Syst.*, volume 21, pages 270–313, New York, NY, USA, Aug. 2003. ACM.

[37] U. Ewaldsson. Cut your network's electricity bill and carbon footprint. `http://wwww.globaltelecomsbusiness.com/Article/2436697/Cut-your-networks-electricity-bill-and-carbon-footprint.html`, February 2010.

[38] W. Fang and L. Peterson. Inter-as traffic patterns and their implications. In *Global Telecommunications Conference. GLOBECOM '99*, pages 1859–1868 vol.3.

[39] A. Feldmann. Internet clean-slate design: what and why? In *SIGCOMM Comput. Commun. Rev.*, volume 37, pages 59–64, New York, NY, USA, July 2007. ACM.

[40] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. In *IEEE/ACM Transactions on Networking,*, volume 1, pages 397 –413, Aug 1993.

[41] M. A. Franklin and T. Wolf. Power considerations in network processor design. In *Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[42] J. Fu, O. Hagsand, and G. Karlsson. Queuing behavior and packet delays in network processor systems. In *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007, MASCOTS '07*, pages 217–224, 2007.

[43] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers Boston, MA, 2000.

[44] S. D. Goglin, D. Hooper, A. Kumar, and R. Yavatkar. Advanced soft-

ware, framework, tools and languages for the ixp family. In *Intel Technology Journal*, volume 7, pages 64–76, 2003.

[45] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[46] J. Guo, J. Yao, and L. Bhuyan. An efficient packet scheduling algorithm in network processors. In *Proceedings of 24th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2005*, volume 2, pages 807 – 818 vol. 2, March 2005.

[47] L. Guo and I. Matta. The war between mice and elephants. In *Ninth International Conference on Network Protocols*, pages 180–188, 2001.

[48] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM '98*, volume 3, pages 1240–1247 vol.3, 1998.

[49] F. Hao, M. Kodialam, and T. V. Lakshman. Accel-rate: a faster mechanism for memory efficient per-flow traffic estimation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 155–166, New York, NY, USA, 2004. ACM.

[50] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency

scaling in chip-multiprocessors. In *ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED 2007*, pages 38–43, 2007.

[51] S. Herbert and D. Marculescu. Variation-aware dynamic voltage/frequency scaling. In *IEEE 15th International Symposium on High Performance Computer Architecture, HPCA 2009*, pages 301–312, 2009.

[52] X. Hesselbach, R. Fabregat, B. Baran, Y. Donoso, F. Solano, and M. Huerta. Hashing based traffic partitioning in a multicast-multipath mpls network model. In *Proceedings of the 3rd international IFIP/ACM Latin American conference on Networking*, LANC '05, pages 65–71, New York, NY, USA, 2005. ACM.

[53] K. Hinton, J. Baliga, M. Feng, R. Ayre, and R. Tucker. Power consumption and energy efficiency in the internet. In *IEEE Network*, volume 25, pages 6 –12, March-April 2011.

[54] E. Horowitz, S. Shani, and D. Mehta. *Fundamentals of Data Structures in C++*. Silicon Pr, 2 edition, 2006.

[55] X. Huang and T. Wolf. Evaluating dynamic task mapping in network processor runtime systems. In *IEEE Transactions o Parallel and Distributed Systems*, volume 19, pages 1086–1098, 2008.

[56] Intel, Microsoft, and Toshiba. Advanced configuration and power interface specifications. `http://www.intel.com/iam/powermgm/specs.html`, 1996.

[57] M. F. Iqbal, J. Holt, J. H. Ryoo, G. de Veciana, and L. K. John. Flow migration on multicore network processors: Load balancing while minimizing packet reordering. In *Proceedings of the 2013 International Conference on Parallel Processing*, ICPP '13, Lyon, France, 2013. IEEE Computer Society.

[58] M. F. Iqbal and L. K. John. Efficient traffic aware power management in multicore communications processors. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 123–134, New York, NY, USA, 2012. ACM.

[59] M. F. Iqbal and L. K. John. Power and performance analysis of network traffic prediction techniques. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2012*, pages 112–113, 2012.

[60] M. F. Iqbal and L. K. John. Traffic aware power management in communications processors. In *Semiconductor Research Corporation Technology Conference*, TECHCON '12, 2012.

[61] H. Jiang and C. Dovrolis. Why is the internet traffic bursty in short time scales? In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 241–252, New York, NY, USA, 2005. ACM.

[62] L. John and A. Subramanian. Design and performance evaluation of

a cache assist to implement selective caching. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '97*, pages 510 –518, Oct 1997.

[63] S. Kaxiras and G. Keramidas. Ipstash: a set-associative memory approach for efficient ip-lookup. In *Proceedings of 24th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2005*, volume 2, pages 992–1001 vol. 2, 2005.

[64] L. Kencl. *Load Sharing for Multiprocessor Network Nodes.* PhD thesis, EPFL, 2003.

[65] R. Kokku, T. L. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. M. Vin. A case for run-time adaptation in packet processing systems. In *SIGCOMM Comput. Commun. Rev.*, volume 34, pages 107–112, New York, NY, USA, January 2004.

[66] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. In *IEEE Micro*, volume 25, pages 21 – 29, March-April 2005.

[67] G. Koren and A. Rosen. Architecture of a 100-gbps network processor for next generation video networks. In *IEEE 26th Convention of Electrical and Electronics Engineers in Israel, IEEEI 2010*, pages 000286 –000290, Nov. 2010.

[68] J. Kuang and L. Bhuyan. Lata: a latency and throughput-aware packet

processing system. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 36–41, New York, NY, USA, 2010. ACM.

[69] J. Kuang and L. Bhuyan. Optimizing throughput and latency under given power budget for network packet processing. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 2901–2909, Piscataway, NJ, USA, 2010. IEEE Press.

[70] B. K. Lee. Exploiting statically identified ilp for network processor applications. In *International Journal of Computer and Electrical Engineering*, volume 2010, October 2010.

[71] B. K. Lee and L. John. Npbench: a benchmark suite for control plane and data plane applications for network processors. In *Proceedings of 21st International Conference onComputer Design, 2003.*, pages 226–233, 2003.

[72] B. Li, G. Venkatesh, B. Calder, and R. Gupta. Exploiting a computation reuse cache to reduce energy in network processors. In *Proceedings of the First international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'05, pages 251–265, Berlin, Heidelberg, 2005. Springer-Verlag.

[73] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM In-*

ternational Symposium on Microarchitecture, 2009. MICRO-42., pages 469–480, 2009.

[74] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada.*, pages 212–223. IEEE Computer Society, 1980.

[75] Z. Liu, K. Zheng, and B. Liu. Hybrid cache architecture for high speed packet processing. In *Proceedings of the 13th Symposium on High Performance Interconnects*, HOTI '05, pages 67–72, Washington, DC, USA, 2005. IEEE Computer Society.

[76] D. Llorente, K. Karras, M. Meitinger, H. Rauchfuss, T. Wild, and A. Herkersdorf. Accelerating packet buffering and administration in network processors. In *International Symposium on Integrated Circuits,ISIC '07*, pages 373–377, 2007.

[77] J. Lu and J. Wang. Analytical performance analysis of network-processor-based application designs. In *15th International Conference on Computer Communications and Networks, ICCCN 2006*, pages 33–39, 2006.

[78] Y. Lu and B. Prabhakar. Robust counting via counter braids: An error-resilient network measurement architecture. In *IEEE International Conference on Computer Communications, INFOCOM 2009*, pages 522–530, April.

[79] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. Elephanttrap: A low cost device for identifying large flows. In *15th Annual IEEE Symposium on High-Performance Interconnects, 2007*, pages 99–108, Aug.

[80] Y. Luo, J. Yang, L. Bhuyan, and L. Zhao. Nepsim: a network processor simulator with a power evaluation framework. In *IEEE Micro*, volume 24, pages 34–44, 2004.

[81] Y. Luo, J. Yu, J. Yang, and L. Bhuyan. Low power network processor design using clock gating. In *Proceedings of 42nd Design Automation Conference, DAC 2005*, pages 712–715, 2005.

[82] Y. Luo, J. Yu, J. Yang, and L. N. Bhuyan. Conserving network processor power consumption by exploiting traffic variability. In *ACM Trans. Archit. Code Optim.*, volume 4, New York, NY, USA, Mar. 2007. ACM.

[83] G. Magklis, P. Chaparro, J. Gonzalez, and A. Gonzalez. Independent front-end and back-end dynamic voltage scaling for a gals microarchitecture. In *Proceedings of the 2006 International Symposium on Low Power Electronics and Design, ISLPED'06*, pages 49–54, 2006.

[84] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *Computer*, volume 35, pages 50–58, Los Alamitos, CA, USA, Feb. 2002. IEEE Computer Society Press.

[85] A. Mallik and G. Memik. A case for clumsy packet processors. In *37th*

International Symposium on Microarchitecture, MICRO-37 2004, pages 147–156, 2004.

[86] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. In *SIGARCH Comput. Archit. News*, volume 33, pages 92–99, New York, NY, USA, Nov. 2005. ACM.

[87] G. Memik and W. H. Mangione-Smith. Increasing power efficiency of multi-core network processors through data filtering. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, pages 108–116, New York, NY, USA, 2002. ACM.

[88] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: a benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, pages 39–42, Piscataway, NJ, USA, 2001. IEEE Press.

[89] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. In *IEEE Micro*, volume 28, pages 69–79, Los Alamitos, CA, USA, Jan. 2008. IEEE Computer Society Press.

[90] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation.

In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 323–336, Berkeley, CA, USA, 2008. USENIX Association.

[91] R. Ohlendorf, M. Meitinger, T. Wild, and A. Herkersdorf. An application-aware load balancing strategy for network processors. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 156–170, Berlin, Heidelberg, 2010. Springer-Verlag.

[92] I. Papaefstathiou, T. Orphanoudakis, G. Kornaros, C. Kachris, I. Mavroidis, and A. Nikologiannis. Queue management in network processors. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 3*, DATE '05, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[93] V. Paxson. End-to-end internet packet dynamics. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '97, pages 139–152, New York, NY, USA, 1997. ACM.

[94] A. Raghunath, A. Kunze, E. J. Johnson, and V. Balakrishnan. Framework for supporting multi-service edge packet processing on network processors. In *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, ANCS '05, pages 163–171, New York, NY, USA, 2005. ACM.

[95] R. Ramaswamy and T. Wolf. Packetbench: a tool for workload characterization of network processing. In *IEEE International Workshop on Workload Characterization, 2003, WWC-6*, pages 42–50, 2003.

[96] B. Sanso and H. Mellah. On reliability, performance and internet power consumption. In *7th International Workshop on Design of Reliable Communication Networks, DRCN 2009*, pages 259 –264, Oct. 2009.

[97] A. Satheesh, D. Kumar, and S. Krishnaveni. Dynamic adaptive self-configurable network processor. In *Proceedings of the 2010 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, UIC-ATC '10, pages 160–164, Washington, DC, USA, 2010. IEEE Computer Society.

[98] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown. Analysis of a statistics counter architecture. In *Proceedings of the The Ninth Symposium on High Performance Interconnects*, HOTI '01, pages 107–, Washington, DC, USA, 2001. IEEE Computer Society.

[99] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. Np-click: A productive software development approach for network processors. In *IEEE Micro*, volume 24, pages 45–54, Los Alamitos, CA, USA, Sept. 2004. IEEE Computer Society Press.

[100] L. Shi, Y. Zhang, J. Yu, B. Xu, B. Liu, and J. Li. On the extreme parallelism inside next-generation network processors. In *26th IEEE Inter-*

national Conference on Computer Communications, INFOCOM 2007, pages 1379 –1387, May 2007.

[101] W. Shi and L. Kencl. Sequence-preserving adaptive load balancers. In *ACM/IEEE Symposium on Architecture for Networking and Communications systems, ANCS 2006*, pages 143 –152, Dec. 2006.

[102] W. Shi, M. MacGregor, and P. Gburzynski. Load balancing for parallel forwarding. In *IEEE/ACM Transactions on Networking,*, volume 13, pages 790 – 801, Aug. 2005.

[103] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of Design Automation Conference, DAC 2001*, pages 524–529, 2001.

[104] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur. Multiprocessor scheduling in processor-based router platforms: Issues and ideas. In *Proceedings of the 2nd Workshop on Network Processors*, pages 48–62, 2003.

[105] A. V. D. Ven. Absolute Power. `http://software.intel.com/sites/oss/pdfs/absolutepower.pdf`.

[106] T. Wolf and M. A. Franklin. Locality-aware predictive scheduling of network processors. In *Proc. of IEEE International Symposium on Per-*

formance Analysis of Systems and Software, ISPASS '01, pages 152–159, 2001.

[107] Q. Wu and T. Wolf. On runtime management in multi-core packet processing systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 69–78, New York, NY, USA, 2008. ACM.

[108] Q. Wu and T. Wolf. Runtime resource allocation in multi-core packet processing systems. In *Proceedings of the 15th international conference on High Performance Switching and Routing*, HPSR'09, pages 62–69, Piscataway, NJ, USA, 2009. IEEE Press.

[109] M. Zadnik and M. Canini. Evolution of cache replacement policies to track heavy-hitter flows. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, pages 31:1–31:2, New York, NY, USA, 2010. ACM.

[110] M. Zadnik, M. Canini, A. Moore, D. Miller, and W. Li. Tracking elephant flows in internet backbone traffic with an fpga-based cache. In *International Conference on Field Programmable Logic and Applications,FPL 2009*, pages 640–644, 2009-Sept.

[111] F. Zane, G. Narlikar, and A. Basu. Coolcams: power-efficient tcams for forwarding engines. In *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications, INFOCOM 2003*, volume 1, pages 42–52 vol.1, 2003.

[112] D. Zhang, Y. Manolopoulos, Y. Theodoridis, and V. Tsotras. Liner hashing. //http://cgi.di.uoa.gr/~ad/MDE515/e_ds_linearhashing.pdf.

[113] W. Zhou, C. Lin, Y. Li, and Z. Tan. Queue management for qos provision build on network processor. In *Proceedings of the Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, FTDCS 2003*, pages 219–224, 2003.