View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

LOAD SHARING FOR MULTIPROCESSOR NETWORK NODES

THÈSE N^{O} 2725 (2003)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Lukas KENCL

Master en Informatique, Charles University, Prague, République Tchèque
et de nationalité tchèque

acceptée sur proposition du jury:

Prof. J.-Y. Le Boudec, directeur de thèse
Dr P. Droz, rapporteur
Dr T. Przygienda, rapporteur
Dr J. Rexford, rapporteur
Prof. P. Thiran, rapporteur

Lausanne, EPFL
2003

© Copyright by Lukas Kencl 2003
All Rights Reserved

Abstract

This thesis discusses techniques for sharing the processing load among multiple processing units within systems that act as nodes in a data communications network.

Load-sharing techniques have been explored in the field of computer science for many years and their benefits are well known, including better utilization of processing capacity and enhanced system fault tolerance. We discuss deploying such methods in the specifics of the networking environment. We concentrate particularly on the data plane, or the data packet-processing tasks. After reviewing the main results in the fields of load sharing and multiprocessor networking systems architectures, we conduct a preparatory optimization study of a router system to gain better understanding of the optimization issues in a particular multiprocessor system.

The main contribution of this thesis, the *adaptive load-sharing method*, is presented next. We first formulate the optimization problem of mapping packets to processors. The goal is to minimize the likelihood of flow reordering, while respecting certain system constraints, such as the acceptable probability of a packet loss. As we show that the task is an $\mathcal{NP}$-complete problem, we propose a heuristic method that uses an adaptive hash-based mapping to assign packets to processors. We demonstrate its advantages and prove that the method adaptation policy possesses the key *minimal disruption property* with respect to the mapping. In other words, the adaptation results in a minimum number of flows being moved among processing units. Further on, the method is validated in an extensive set of simulations designed to imitate the networking environment.

Finally, two sample applications, an architecture of a multiprotocol router and an implementation of a server load balancer on a network processor demonstrate the applicability of the method.

Zusammenfassung

Thema der vorliegenden Doktorarbeit sind Techniken zur Lastverteilung zwischen Prozessoren in Systemen, welche die Funktion von Netzwerkknoten in einem Datenkommunikationsnetz haben.

Techniken zur Lastverteilung werden in der Informatik schon seit vielen Jahren erforscht, und ihre Vorzüge, wie bessere Ausnutzung der Rechnerkapazität und eine grössere Fehlertoleranz des System, sind bestens bekannt. Die vorliegende Arbeit befasst sich mit der Anwendung dieser Methoden unter den spezifischen Bedingungen von Datennetzen. Hierbei liegt der Schwerpunkt der Betrachtung auf der Datenebene, bzw. auf der Verarbeitung von Datenpakten.

Nach einer Übersicht zu den wichtigsten Resultaten aus den Bereichen Lastverteilung und Architekturen von Multiprozessor-Netzwerksystemen wird eine erste Optimierungsstudie eines Router-Systems erarbeitet. Ziel ist hier die Erlangung eines tieferen Verständnisses der Optimierungsproblematik am Beispiel eines ausgewählten Multiprozessorsystems.

Anschliessend wird der Hauptbeitrag der vorliegenden Doktorarbeit, die *Methode adaptiver Lastverteilung*, vorgestellt. Zuerst wird das Optimierungsproblem für die Verteilung von Paketströmen auf Prozessoren formuliert. Ziel hierbei ist die Minimierung der akzeptablen Wahrscheinlichkeit einer ungewünschten Neuzuordnung eines Datenstromes bei Berücksichtigung bestimmter Systemvorgaben, wie z.B. der Wahrscheinlichkeit eines Paketverlustes. Da im folgenden gezeigt wird, dass diese Aufgabe ein $\mathcal{NP}$–vollständiges Problem ist, wird eine heuristische Methode vorgeschlagen, welche ein adaptives, hash-basiertes Verfahren für die Verteilung der Pakete auf die Prozessoren verwendet. Die Vorzüge dieser Methode werden aufgezeigt und

es wird bewiesen, dass das Adaptionsverfahren die grundlegende Eigenschaft minimaler, durch Neuzuordnung verursachter Paketstromunterbrechungen (*minimal disruption property*) besitzt. Damit verursacht der Adaptionsvorgang die Verschiebung einer lediglich minimalen Anzahl von Paketströmen zwischen den Prozessoren. Anschliessend wird die Methode in umfassenden Simulationen validiert. Dabei werden typische Einsatzbedingungen modelliert.

Die Anwendbarkeit der Methode wird anhand zweier Beispiele – der Architektur eines Multiprotokoll-Routers und der Netzwerkprozessor-Implementierung einer Lastverteilung für eine Serverfarm – unter Beweis gestellt.

To Jitka.

Acknowledgments

First of all, it is a great pleasure to thank my advisor, Prof. Jean-Yves Le Boudec, who has provided me with excellent guidance throughout my Ph.D. research and has had enormous patience with me. I thank him for the many fruitful discussions and his vital contributions to this thesis.

I also thank the IBM Zurich Research Laboratory for providing me with a splendid working environment. It is a delight to work in a modern environment and with such competent and creative people. I am especially grateful to my manager, Patrick Droz, who has supported me throughout, helped me narrow the scope of the thesis by asking the right questions and made sure I had enough time and energy to finish it.

Furthermore, I thank my colleagues, namely Daniel Bauer, Ed Bowen, Robert Haas, Ton Engbersen, Laurent Frélechoux, Mitch Gusat, Andreas Herkersdorf, Ilias Iliadis, Andreas Kind, Bernard Metzler, Cyriel Minkenberg, Sean Rooney, Patricia Sagmeister, Jan Van Lunteren, Marcel Waldvogel and many others, for providing me with excellent comments and pointers to relevant publications and for creating a friendly working atmosphere. Special thanks go to my office mates, Sonja Buchegger and Roman Pletka, who not only spent endless time in scientific discussions with me, but were also enormous fun.

I also thank our visiting students, Bozidar Radunovic and Riccardo Russo, for their creative approach to the problems studied and their significant contributions to exploring the various subtopics of this work.

My sincere thanks to Lilli-Marie Pavka, Charlotte Bolliger and Urs Bitterli for helping me improve the quality of my publications enormously.

Last, but certainly not least, I thank my wife, Jitka. It is only by virtue of her love and sacrifice that I was able to finish this work. Being a wonderful wife and a caring mother, she has made the years spent working on this thesis fine and enjoyable.

"However, how to distribute packets across parallel links while achieving load balance among the links and maintaining packet order in the same flow remains challenging."

H. Jonathan Chao, *Next Generation Routers*, September 2002.

Contents

Abstract iii

Zusammenfassung v

Acknowledgments ix

1 Introduction 1

1.1 Optimizing network nodes 1

1.2 Outline 3

1.3 Claims 4

2 State of the art 7

2.1 Introduction 7

2.2 Load sharing and networking 7

2.2.1 Load sharing in general 7

2.2.2 Load sharing in networking 8

2.2.3 Robust hash routing 10

2.3 Router architecture 14

2.4 Server farms 18

2.4.1 Architectural demands 18

2.4.2 Web caching and content delivery networks 19

2.4.3 Server farm load balancing 20

2.5 Conclusions 24

3 Optimizing router architecture **27**
3.1 Introduction . . . 27
3.2 General distributed router architecture . . . 28
3.2.1 Router model . . . 28
3.2.2 LFE and MFE model . . . 30
3.2.3 Switch model . . . 33
3.2.4 Time a packet spends within the system . . . 37
3.3 Cost optimization . . . 37
3.3.1 Forwarding engine cost . . . 37
3.3.2 Switch cost . . . 37
3.3.3 Total system cost . . . 39
3.3.4 Optimization problem . . . 39
3.4 Numerical results . . . 39
3.4.1 Optimization . . . 39
3.4.2 Total system cost . . . 40
3.4.3 Distribution of resources—LFEs, MFEs and switch capacity . 41
3.4.4 Distribution of processing capacity—LFEs and MFEs . . . 42
3.4.5 Switch speed . . . 45
3.5 Load sharing in a router . . . 47
3.5.1 Acceptable load sharing . . . 47
3.5.2 Other system requirements . . . 48
3.6 Conclusions . . . 49

4 Adaptive load sharing **51**
4.1 Introduction . . . 51
4.2 Notation and assumptions . . . 52
4.3 Packet-to-NPU mapping . . . 54
4.4 Optimization problem statement . . . 57
4.5 Adaptive load-sharing heuristic . . . 59
4.5.1 Adaptation algorithm . . . 59
4.5.2 Triggering policy . . . 60

4.5.3 Minimal disruption 63
4.5.4 Adaptation policy 67
4.6 Conclusions 69

5 Method validation 71
5.1 Introduction 71
5.2 Simple simulator—proof of concept 72
5.2.1 Potentially congested case 74
5.2.2 Wrong initial weights case 76
5.3 Realistic traffic generation 76
5.4 Router system model 80
5.5 Results 81
5.5.1 Adaptive load sharing 81
5.5.2 Influence of maximal flow rate ϵ_f 86
5.5.3 Fractional factorial analysis of the load-sharing method . . . 87
5.5.4 Influence of the number of processors 89
5.6 Conclusions 91

6 Applications in networking systems 93
6.1 Introduction 93
6.2 Multiprotocol router 94
6.2.1 Router architecture 94
6.2.2 Load indicator 96
6.2.3 Pseudo-random function 96
6.3 Server load balancer on a network processor 100
6.3.1 Server farm topology 100
6.3.2 Load Balancer on the PowerNP network processor–
–general overview 103
6.3.3 Network processor data plane 105
6.3.4 HRW weights representation on the network processor 107
6.3.5 HRW weights data structure 111
6.3.6 Network processor data plane implementation 118

6.4 Conclusions 119

7 Conclusions 121

7.1 Open issues 121

7.2 Concluding remarks 122

Bibliography 123

A List of publications and Curriculum Vitae 131

List of Tables

5.1 Acceptable load sharing. 83
5.2 Flow remappings. 85
5.3 Fractional factorial analysis results. 87
5.4 Effects of factors. 88
5.5 Influence of the number of NPUs. 91

6.1 Kernel IP routing table 101
6.2 Kernel ARP table 102
6.3 Results of the simulation using a flat data structure. 114
6.4 Results of two simulations on hierarchical data structure 117

List of Figures

2.1 Mapping disruption problem 11
2.2 Robust hash routing minimizes the mapping disruption. 13
2.3 Distributed router architecture 16
2.4 Parallel router architecture 17
2.5 Server load balancing topology. 20

3.1 LFEs, switch and MFEs within the general distributed router architecture model. 29
3.2 Model of the general distributed router architecture with multiple LFEs and MFEs. 30
3.3 Traffic flows within the switch 35
3.4 Total system cost. 41
3.5 Fraction of traffic enqueued at LFEs, r. 42
3.6 Total processing power of the FEs. 43
3.7 Total cost of the FEs. 43
3.8 Switch saturation throughput. 43
3.9 Switch cost. 44
3.10 Optimal processing power per individual LFE. 44
3.11 Total processing power of the LFEs (note that the graph was adjusted for the reader's convenience in order to be able to depict the values equal to 0, which would normally tend to negative infinity on a logarithmic graph). 44
3.12 Total processing power of the MFEs. 45
3.13 Fraction of LFE processing power out of total system processing power. 45

3.14 Switch port transmission speed. 46
3.15 Traffic at the switch master port—the header passing overhead within the switch. 46
3.16 Fraction of the header passing overhead out of the total switch traffic. 46

4.1 Load-sharing scheme abstraction 55
4.2 Load sharing with feedback 60
4.3 Adaptation algorithm. 61
4.4 Example of the dynamic workload intensity threshold as a function of a hypothetical filtered total system workload intensity. 62
4.5 Example of the hysteresis bound as a function of a hypothetical filtered total system workload intensity. 63
4.6 Example of the triggering threshold as a function of a hypothetical filtered total system workload intensity. 64
4.7 Minimal disruption property of the adaptation. 68

5.1 The simple load-sharing system simulator. 72
5.2 Number of packets at the ingress queue 75
5.3 Evolution of the weights' value 75
5.4 Total number of packets queued in the output queues 75
5.5 System workload intensity 75
5.6 Standard deviation 75
5.7 Number of packets at the input queue. 77
5.8 The evolution of the weights' value. 77
5.9 Queue occupancy. 77
5.10 Link workload intensity. 77
5.11 Standard deviation function. 77
5.12 Flow length cumulative distribution. 78
5.13 Per-processor workload intensity 82
5.14 Packets dropped 83
5.15 Triggering and adaptation policy. 84
5.16 Flows re-mapped 85

5.17 Influence of maximal per-flow fraction rate limit ϵ_f 86
5.18 Influence of the number of NPUs 90

6.1 Load sharing within a distributed multiprotocol router 94
6.2 Multiprotocol router, consisting of an input- and output switch/shared memory and a pool of multiple NPUs, sharing the load of N line cards 95
6.3 Sample server farm topology 102
6.4 Server load balancer on the PowerNP network processor. 104
6.5 NP state diagram 106
6.6 Ross' weights distribution. 109
6.7 Russo's weights distribution. 110
6.8 HRW Mapping computation algorithm 112
6.9 Flat data structure. 113
6.10 Hierarchical data structure 115
6.11 Topology for the hierarchical data structure simulations 117

Chapter 1

Introduction

1.1 Optimizing network nodes

With the ever-increasing transport capacity of links of complex data networks such as the Internet, it is the various processing nodes along the paths that the data traverses that are becoming the bottlenecks, preventing speedy and reliable data delivery.

As we are unable to raise the processing capacity of individual processors at an adequate rate, solutions on the device system level that aggregate capacities of multiple processors are increasingly being designed to support the growing data rates.

The functionality requirements for such network elements (routers, servers, server farms, proxies, caches, load balancers, firewalls) are becoming increasingly complex. With respect to networking, we typically distinguish between two kinds of functionalities associated with data communication:

- the *control plane* functionality, which exercises the overall control of the networking communication and the signaling among nodes, and
- the *data plane* functionality, which exercises the actual forwarding, processing, alterations or manipulations of data packets.

The work presented in this thesis concentrates on optimizing the *data plane* functionality of network elements, or, more specifically, optimizing the data plane functionality of network nodes that consist of *multiple processing units*.

The key contribution of this work is a *method that optimizes the way the processing load is shared among the multiple processors*, while respecting the specific constraints of the networking environment.

We assume that data for processing arrives over network links in the form of packets and that individual packets belonging to a specific connection between end hosts form a flow. Thus, traffic over each network link consists of packets belonging to various flows.

For several reasons, it is desirable that packets belonging to the same flow be processed by the same processor. Processing at different processors may result in packet reordering within a flow. Furthermore, a number of networking applications requires that packets from one flow be processed within some context—for example for reasons of policing or shaping, or in order to process subsequent requests from the same host. The information carried in each packet may require various amounts of processing power and the flows may consist of uneven amounts of packets. The method we present aims to *fulfill the flow-preservation requirement while keeping the load on the multiple processors within some degree of balance.* Furthermore, the decision where to process a packet is reached *fast and without maintaining state information on the flows.*

There are several factors that contribute to the difficulty of solving this task. There is high variation in network traffic patterns and they are not easily predictable. Furthermore, even if knowledge of the near future were available, the task of mapping various flows to processors and keeping the load balanced, while optimizing a system-wide function such as the amount of packets reordered, is $\mathcal{NP}$-Complete.

The advantages of deploying such a load-sharing method are manifold:

- *It increases the total processing capacity of the system.* The total load of the system is distributed over the processing elements in an optimal way and thus its processing capacity can be fully utilized;
- *It increases the system flexibility.* Addition or removal of processing capacity can be performed in a seamless manner with minimal disruption to the processing system;

- *Scalability.* The method ensures that increasing the processing capacity of the system results in equivalent increase in processing performance of the system;
- *Fault tolerance.* A failure in one of the processing nodes is seamlessly hidden by others assuming its task (sharing its load).

There are many applications or systems in the networking area, for which deploying such a method is beneficial. We present two examples, a router and a server farm, but any system requiring that the packet processing load be spread over multiple processing units or destinations would benefit from the method.

The main contribution of this work is the presented load-sharing method. It is an *adaptive heuristic*: adaptive to cope with the varying traffic patterns, and a heuristic as it estimates the near-optimal solution within a very short time.

The method is hash-based, extending the robust hash routing method by Ross [50]. The fact that it is hash-based means that the mapping decision is achieved fast and that no state information on individual flows needs to be stored.

As the robust hash routing method leads to perfect load balancing over non-biased traffic patterns, we introduce the adaptive control loop to cope with the biased patterns. The key contribution of this work is the adaptation policy—*we prove that the proposed adaptation policy possesses the minimal disruption property with respect to the flow-to-processor mapping*—meaning that only a minimum amount of flows is moving among processors when the mapping is adjusted.

Furthermore, we present an extension of the method that shows how, by maintaining little state information, mapping disruption can be avoided altogether.

1.2 Outline

In Chapter 2, we present an overview of the load-sharing methods in general and of the specific solutions to load-sharing problems encountered in networking. Furthermore, we review the state-of-the-art in the two domains considered for deployment of the load-sharing method, router architecture and server load balancing.

In Chapter 3, a preparatory study of a router architecture is presented, to provide a better understanding of the router design space and of the potential benefits of deploying a load-sharing method within a router.

The main contributions of this thesis are presented in Chapter 4. Here, the *adaptive load-sharing* method is described in detail and the *minimal disruption property* is proved by theoretical means.

In Chapter 5, a practical validation of the method is performed. First, a small, Java-based implementation is used as a proof of concept, and than a large, Matlab-based set of simulations to explore various parameters of the method over Internet-like generated traffic is carried out and discussed.

Real-world system applications of the method are discussed in Chapter 6. An existing implementation of a server farm load balancer on an IBM PowerNP network processor is described in detail, as well as potential implementations in a router system. Some method implementation issues, such as the optimal data structure for the mapping weights, are reviewed.

Finally, in Chapter 7, we offer some concluding remarks on the future applicability of the method and related open issues.

1.3 Claims

- A new method for providing adaptive load sharing among multiple processing units in a networked environment is presented. The method is hash-based, requires a minimum of state information to be maintained and leads to a minimum of packet flow disruptions. The method comprises a flow-to-processor mapping and an adaptive feedback mechanism;

- The minimal disruption property is proved by theoretical means;

- Model of a router equipped with the load-sharing method has been implemented in MATLAB;

- A traffic generator for generating realistic Internet traffic patterns has been implemented in MATLAB. The method has been tested extensively using the generated traffic;

- An extension of the method, which leads to zero flow disruption by adding a small amount of state information and duplicating the hash computation, has been developed;
- A prototype of the method has been implemented on the IBM PowerNP network processor to act as a Web server load balancer;
- A study of optimizing the router architecture has been conducted, concluding that parallelism and load sharing on the router data path are applicable and advantageous;
- Two patent applications have been submitted on the two versions of the load-sharing method.

Chapter 2

State of the art

2.1 Introduction

In this chapter we are going to review the previous fundamental results in fields related to the topic of this thesis.

Load sharing or load balancing problems have been extensively studied in computer science for many years and there exists a number of works in this area. We present those that were crucial for our work and in the final part of the load sharing discussion we examine in detail the robust hash routing technique, which forms the basis of our optimized load-sharing method.

In the second and third sections of the chapter, we examine more closely previous works on the two classes of network nodes that we discuss in more detail throughout the thesis—multiprotocol routers and Web server farms.

2.2 Load sharing and networking

2.2.1 Load sharing in general

For a general survey of load-sharing algorithms, see [55]. A widely accepted taxonomy of load-sharing algorithms has been presented by Casavant and Kuhl [8].

Eager, Lazowska and Zahorjan [17] have studied specific adaptive load-sharing policies, consisting of a transfer and a location policy. Their work shows that simple adaptive load-sharing policies yield significant performance improvements relative to the no load-sharing case and, at the same time, performance very close to complex adaptive policies. In addition, a threshold-based location policy is shown to bring substantial improvements over a random selection location policy.

The task of determining a processing unit on which a specific processing job should be executed so that a system-wide function is optimized has been shown to be $\mathcal{NP}$-complete in general. El-Rewini, Ali and Lewis [19] provide an overview of the $\mathcal{NP}$-completeness proofs for various instances of the problem and discuss some candidate heuristics for the solutions.

A heuristic, producing an answer which is not necessarily optimal, but is achieved in short time, is typically used. Such a global task scheduling heuristic usually takes some kind of dynamic processor workload information as input. The most effective representation of the workload index has been a topic of intensive research. Kunz [35] has demonstrated that a single, one-dimensional workload descriptor yields better results than more complex descriptors.

2.2.2 Load sharing in networking

Load-sharing methods have recently been studied in relation to the task of distributing Internet traffic over multiple links or paths within the network [7], [6] [53]. Cao, Wang and Zegura evaluate in [7] the performance of various fast static hashing schemes, as well as of one adaptive, for splitting traffic among multiple links. The following static hash functions are studied:

1. Modulo over Destination Address

$$H(\cdot) = DestIP \ mod \ N.$$

2. XOR Folding of Destination Address:

$$H(\cdot) = (D_1 \oplus D_2 \oplus D_3 \oplus D_4) \ mod \ N,$$

where D_i is the i-th octet of the destination IP address

3. XOR Folding of Source and Destination Addresses:

$$H(\cdot) = (S_1 \oplus S_2 \oplus S_3 \oplus S_4 + D_1 \oplus D_2 \oplus D_3 \oplus D_4)\ mod\ N,$$

   where S_i and D_i are the i-th octets of the source and destination IP addresses, respectively.

4. Internet Checksum of the TCP 5-tuple (source and destination IP address, source and destination port, and protocol ID):

$$H(\cdot) = CheckSum(5 - tuple)\ mod\ N.$$

5. The 16-bit Cyclic Redundancy Check (CRC) function:

$$H(\cdot) = CRC16(5 - tuple)\ mod\ N.$$

The study concludes that except for the CRC-based hash, other methods result in relatively poor traffic load-balancing.

Furthermore, the authors study an adaptive, table-based method, which, when adapted according to the monitored load, gives comparable performance as the CRC-based function, even when using the XOR folding as the primary indexing technique into the table. However, note that such method, as presented, requires to maintain considerable state information on the mapping of table bins. Furthermore, the adaptations may lead to significant disruptions in the mapping.

Basturk et al. [6] explore the possibilities of using the IP anycast routing to distribute traffic load among multiple paths or servers. Several load distribution techniques that pin packets from a particular flow to certain route are discussed. Four different disciplines for selecting the route are evaluated on real network traffic traces: round-robin, random, hash-based and least connections. An interesting finding with respect to the work presented in this thesis is that all the disciplines perform approximately equally well in the key metric of distributing the number of bytes transferred among the multiple destinations.

Shaikh, Rexford and Shin [53] concentrate on the problem of mapping traffic flows onto multiple network paths in order to achieve better bandwidth utilization and routing stability. The method divides traffic flows into short-lived and long-lived and uses different mapping disciplines for each group, an adaptive one for the long-lived and a static one for the short-lived flows. It is demonstrated that for this problem, thanks to the particular length distribution of network flows, such a hybrid approach is beneficial over each method stand-alone, achieving better balance on one hand and saving on signalling overhead on the other. The study of flow length distribution in [53] has been inspirational for some of the experiments presented in Chapter 5.

Particular interest in load sharing has recently been raised in the areas of Web servers, Web caching and clustered digital libraries [5], [24], [50], [69].

2.2.3 Robust hash routing

The robust hash routing (also known as Cache Array Routing Protocol (CARP)) distributed caching scheme, which uses the highest random weight (HRW) algorithm developed by Thaler and Ravishankar [62], and its more fine-grained, weighted version by Ross [50], is a popular choice for Web caches and is implemented in products offered by Microsoft [5]. This algorithm has become a foundation for our work.

The following example illustrates the general mapping disruption problem, the flaws of basic mapping functions and subsequently the advantages provided by the robust hash routing.

Let $h(\cdot)$ be a hash function that maps the space of all n-bit binary numbers vectors $\vec{v}$ to a hash space H. The hash space H is partitioned into N consecutive intervals, corresponding to N destinations for the mapped objects. Assume that a new destination is added to the N existing ones and thus there are now $N+1$ possible destinations. What fraction of objects do change their destination, if the same mapping (hash) function is still being used? Thaler and Ravishankar [62] refer to this fraction as the *disruption coefficient*.

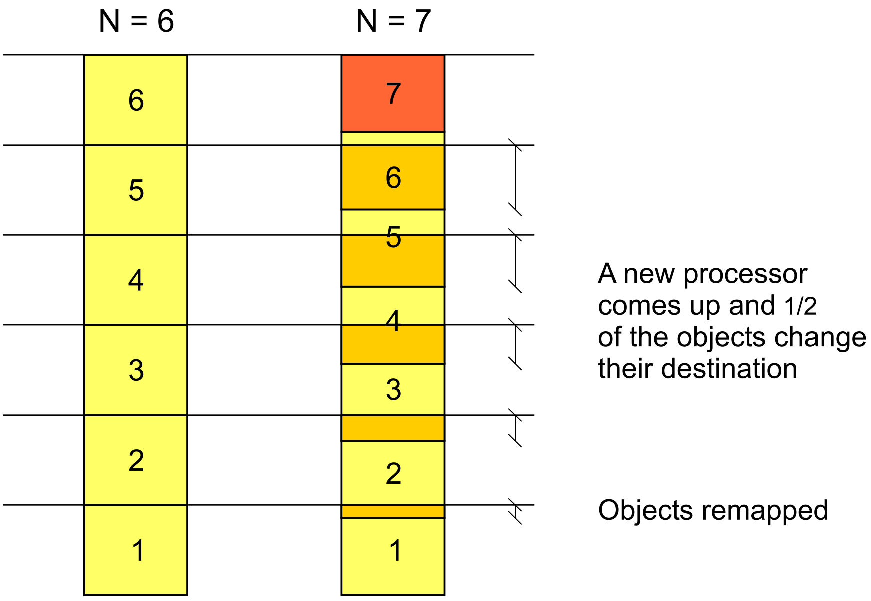

N = 6
N = 7
6
5
4
3
2
1
7
6
5
4
3
2
1
A new processor comes up and 1/2 of the objects change their destination
Objects remapped

Figure 2.1: Mapping disruption problem

Initially, the hash space is divided into N sets as follows:

$$\left[0, \frac{1}{N}\right], \left[\frac{1}{N}, \frac{2}{N}\right], \ \ldots \ , \left[\frac{N-1}{N}, 1\right].$$

If the hash function produces the score $sc \in \left[\frac{j}{N}, \frac{j+1}{N}\right]$, then destination j will be chosen. Suppose that all destinations are equally probable to be mapped to, meaning that the distribution of sc is uniform over the hash space H. Then, the hash space will be divided, upon the addition of the new destination, as follows (see Fig. 2.1):

$$\left[0, \frac{1}{N+1}\right], \left[\frac{1}{N+1}, \frac{2}{N+1}\right], \ \ldots \ , \left[\frac{N-1}{N+1}, \frac{N}{N+1}\right], \left[\frac{N}{N+1}, 1\right].$$

Only flows that belong to the following subsets have not changed their destination:

$$\left[0, \frac{1}{N+1}\right], \left[\frac{1}{N}, \frac{2}{N+1}\right], \left[\frac{2}{N}, \frac{3}{N+1}\right], \ \ldots \ , \left[\frac{N-1}{N}, \frac{N}{N+1}\right].$$

The sum of these intervals represents the disruption coefficient:

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

This means that in this case 50% of the mapped objects have changed destination. Thaler and Ravishankar prove that the disruption coefficient ranges between $1/N$ and 1 for all mappings.

The robust hash routing is a mapping designed to avoid large disruption coefficient. With the *robust hash routing* [62], the object identifier $\vec{v}$ and the destination index j are used together to generate a hash value or score. For every destination j, a score $h(\vec{v}, j)$ is computed and the final destination chosen is the one with the maximum score:

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ h(\vec{v}, j) &= \max_{k \,\in\, [1,N]} h(\overrightarrow{v}, k). \end{aligned} \tag{2.1}$$

Thaler and Ravishankar [62] demonstrate that if a new destination is added, only a fraction of $\frac{1}{N+1}$ of the objects is re-mapped (see Fig. 2.2), which is the minimum amount. Thus, we say that the HRW mapping possesses the *minimal disruption* property.

In the form above, the robust hash routing guarantees balancing of the mapped objects over the destinations, but only in case of homogenous size of the destinations. Ross [50] adds multiplicative weights into Eq. 2.2 to provide for heterogenous load distributions over destinations:

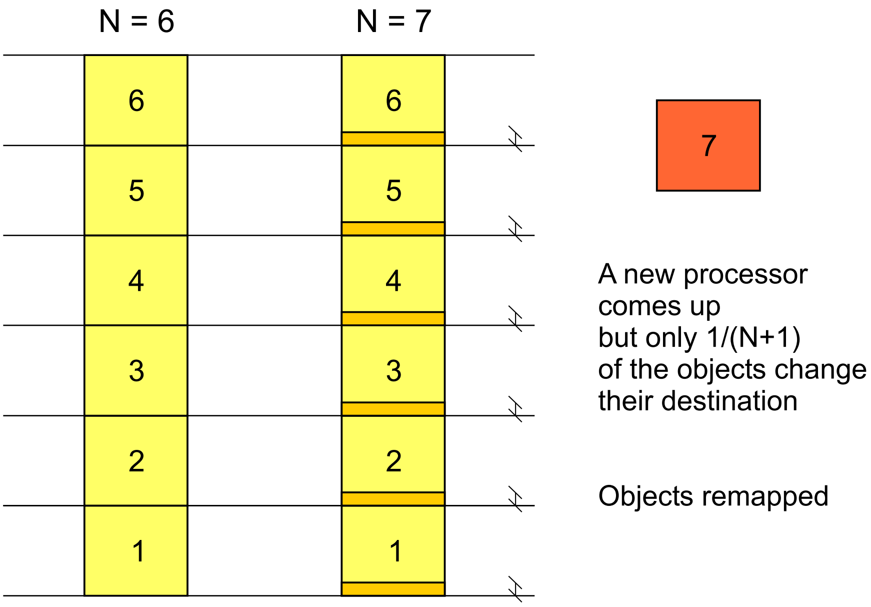


Figure 2.2: Robust hash routing minimizes the mapping disruption.

Def. 1 ***HRW Mapping*** ***f*.** *Let* $h(\vec{v}, j)$ *be a pseudo-random function* $h : V \times \{1, 2, \ldots, m\} \rightarrow (0, 1)$*, i.e., we assume* $h(\vec{v}, j)$ *to be a random variable in* $(0, 1)$ *with uniform distribution. The HRW mapping* $f(\vec{v})$ *is then computed as follows:*

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ x_j \cdot h(\vec{v}, j) &= \max_{k \in [1,N]} x_k \cdot h(\vec{v}, k) \end{aligned} \tag{2.2}$$

where $x_j \in \mathbb{R}^+$ *is a weight multiplier assigned to each destination.*

Ross [50] also introduces a formula for computing the weights x_i, depending on the target probability p_i of each destination:

Theorem 1 (Ross) *Let* $p_1, \ldots, p_N$ *be given target probabilities. Reorder the destinations so that* $p_1 \leq \ldots \leq p_N$*. Let*

$$x_1 = (Np_1)^{1/N}$$

and let $x_2, \ldots, x_N$ *be calculated recursively as follows:*

$$x_n = \left[\frac{(N-n+1)(p_n - p_{n-1})}{\prod_{i=1}^{n-1} x_i} + x_{n-1}^{N-n+1} \right]^{\frac{1}{N-n+1}}. \tag{2.3}$$

Then the robust hash routing algorithm with multipliers $x_1, \ldots, x_N$ *will map the fraction* p_n *of incoming objects to the* n*-th destination,* $n = 1, \ldots, N$.

To summarize, the HRW mapping possesses the following significant advantages over other hash-based load balancing schemes [50] [62]:

Load balancing The mapping provides load balancing over the request object space, even for the heterogeneous case. It allows to split the hashed objects into hash buckets of arbitrary size, as determined by predefined weights.

Minimal disruption In case of a processor failure, removal or addition, the number of request objects that are re-mapped to another destination is minimal.

2.3 Router architecture

Latest developments in transmission technologies have led to an enormous increase in the amount of data transported over the links of the Internet. Such a rapid evolution places significant strain on the interconnecting equipment, primarily routers, to scale with the pace of the transmission speed increase. Recent works [21], [34] have provided a basis for the new generation of interconnecting devices by presenting the first gigabit and terabit router architectures. These works have built on new developments in the areas of switch architectures [20], [39] and fast lookup algorithms [14], [65], [44].

It is becoming increasingly difficult to satisfy the demands for router performance with a traditional centralized architecture [21]. In the case of multiple router inputs and high throughput, the single central processor is not able to cope with its processing task. In order to eliminate the packet processing bottleneck, *multiple* processing units, known as forwarding engines (FE), or, more sophisticated, network processors

(NPU), are typically deployed in contemporary routers. A router system thus consists of multiple processing units gathered around a switch element. Packet processing within such a system is essentially *distributed*. The previously centralized router devices are thus being replaced by routers of more effective architectures, distributed or parallel [10], [64], [11].

In principle, packet processing can either be carried out directly at router inputs, using local forwarding engines (LFE), or at remote master forwarding engines (MFE), reachable through the switch element. Both of these paradigms may be combined in one system. Given the performance demands, a single MFE may not be sufficient and thus MFEs are often grouped into pools of parallel MFEs. Critical related questions emerge—what are the best capacities, locations and schemes of cooperation of all the elements (switch, LFEs, MFEs) within a router system in order to satisfy given system performance demands, and what is the most economical alternative to satisfy those demands?

In the case of a *distributed architecture* [9], most of the packet processing load is performed by processors typically located directly at the router inputs. Such an architecture has the drawback of poor utilization because all the processors are hardly ever saturated, as the load is almost never evenly distributed over the inputs and does not always reach the nominal rate. *Parallel router architectures* [4], [23] are based on a pool of parallel processors, located remotely from the inputs, with all of the processors being able to perform the data path processing tasks. Packets may be buffered at the inputs, and relevant fields of the packet (for example, the packet header) are being sent to the pool for resolution. Such an architecture does not suffer from underutilization because loads of all the inputs are combined at the pool. Instead, the pool interconnect tends to become a major bottleneck. Another drawback is that if load balancing is performed over the pool, the load balancing device is a single point of failure for the entire router.

Two of the possible router architectures belonging to this space (albeit without considering the switch element), *fully distributed* and *parallel*, were examined and compared in [10]. In the fully distributed case (Fig. 2.3), each line card (LC) has a dedicated LFE attached. When a packet arrives at an LC, its LFE searches for

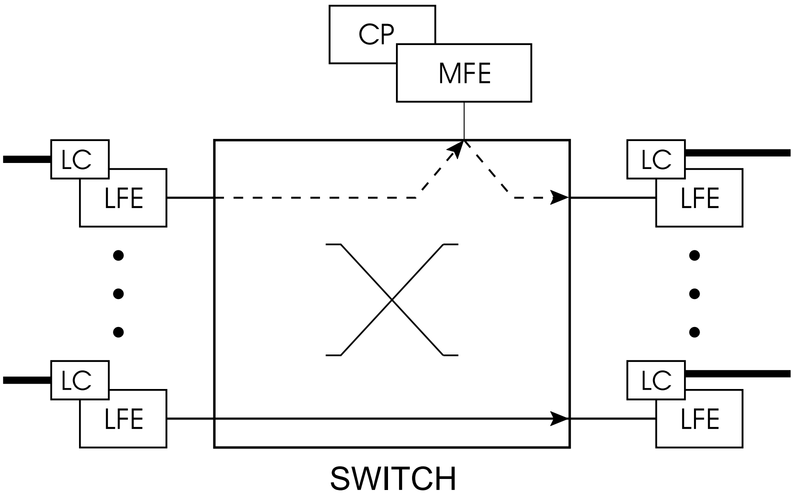


Figure 2.3: Distributed router architecture (LC: line card, LFE: local forwarding engine, MFE: master forwarding engine, CP: control point).

the appropriate route. If the route is found, the packet is immediately forwarded through the switch to the output LC. If an LFE is not able to determine the route (e.g. contains only a part of the routing table), or does not have enough processing power to handle all the arriving packets, it sends the packet header to the MFE, which contains a copy of the entire routing table and therefore is able to find the appropriate route. In general, as the MFE stores the entire routing table and should be able to assist all LCs, it is considerably more powerful than an LFE.

In the case of a parallel router architecture (Fig. 2.4), the router contains a pool of several high-performance MFEs that handle the router's entire workload. Any MFE can take on a new request as soon as it has processed the previous one. As long as the switch can handle the additional traffic, the total system processing power is considerably higher than in the case of a fully distributed system, but also the total system cost rises accordingly.

The content of the routing table is managed by the router control point (CP), which often resides in the same hardware unit as the MFE. The CP uploads the table to the FEs. As the CP is a processor dedicated to the control plane rather than to the data plane within router, it is not considered in the optimization, neither in [10],

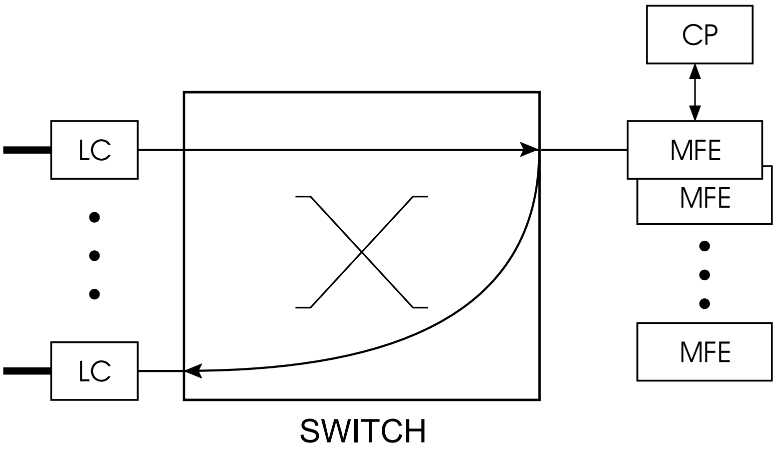


Figure 2.4: Parallel router architecture (LC: line card, MFE: master forwarding engine, CP: control point).

nor in this work.

A simple framework for assessing the cost vs. performance ratio was applied in [10]. The cost and performance differences between a fully distributed and a parallel architecture, as well as the influence of various system parameters on the ratio, were studied. The optimizations were carried out by constraining the maximal packet-processing time and the maximal FE processing power while minimizing the total cost of the system.

The results of the optimization in [10] in the case of the distributed architecture indicate that as the cost ratio between MFE and LFE increases, it is more efficient to use the fully distributed architecture rather than a centralized one without LFEs. Similar behavior occurs when the fraction of packets an LFE unsuccessfully processes decreases. The parallel architecture is more expensive than the distributed one for the same workload, but it is scalable and thus able to handle a much higher workload.

Other designs [21], [51] seek to combine both approaches by containing remotely located (at a different switch port than the input line cards) network processors or forwarding engines, which serve a certain predefined set of inputs to carry out the packet processing tasks on packets arriving at a these inputs. Again, the traffic may not be evenly distributed over these sets, which leads to less efficient utilization.

Tantawy, Zitterbart and Koufopavlou [33] [61] have concentrated on exploring the possibilities of parallel implementations of the TCP/IP packet processing within routers. In these studies, functional decomposition of individual packet processing tasks has been determined and various possible forms of parallelism have been categorized: spatial parallelism, pipelining or concurrent operation.

In a study on future router architectures, Kumar, Lakhsman and Stilliadis [34] emphasize that due to the nature of networking transport protocols, it is often illegal, or at least extremely undesirable, to allow packet reordering within a packet flow. Although the widely used TCP protocol attempts to tackle this problem by correct reordering at the destination, reordering slows down data delivery, increases receiver buffer size and still may not prevent some undesirable retransmissions and subsequent network congestion. If packets from the same flow are to be processed by different processors, packet reordering can easily occur. Therefore, packets belonging to a particular flow should be processed by the same processor.

Dittman and Herkersdorf [16] present a hardware-based load balancer, designed for scaling the performance of network processors that handle the load at router interfaces. A single high-speed link can be balanced over multiple network processors of lower capacity. The load balancer uses an adaptive hash-table. Flows that exceed the capacity of a single processor are sprayed over multiple processors.

2.4 Server farms

2.4.1 Architectural demands

Connection speed, low latency, fault tolerance, and ease of management are the keywords for the contemporary web server architectures. As web sites handle ever-increasing numbers of clients, the traditional solution, increasing the capacity of the servers, is becoming neither economically sound, nor scalable. Various techniques have been proposed and implemented in recent years in order to scale with the growth of the Internet.

The solutions must continue to function appropriately even as they (or their context) evolve in size or volume. Flexible, easy-to-integrate products are required. Fault tolerance is a must, as the web sites must be able to operate not only under any traffic condition (including hostile traffic such as hacker attacks), but also in cases of unexpected internal problems (like software errors or hardware failures).

The typical contemporary solutions addressing the Web growth are server farm load balancing, Web caching and Content Delivery Networks (CDNs). Before reviewing the server farm load balancers in more detail, we provide a general overview of the other two solutions.

2.4.2 Web caching and content delivery networks

Web caching

This technique employs local memories (caches) containing copies of the objects accessed most often (for example web pages). There are typically three ways to implement caching. With *Proxy Cache*, the internet browser is configured to first look for the resource directly in the cache. If the cache does not contain the object the web browser will contact the Web server. This approach is not generally used because it requires manual configuration. With *Transparent Caching*, the network automatically redirects the request to one or more caches through devices called cache re-directors. If a cache does not contain the required content, the request is redirected to the actual web server. Finally, with *Reverse Proxying*, the Web cache receives requests from the clients, proxies them to the Web server, and caches the response itself on its way back to the client. This means that when the request is repeated, the proxy server itself can provide static content from its cache.

Content delivery networks

Content Delivery Networks (CDNs) are private networks of geographically dispersed caching servers at the edge of the Internet. They bring content (like, for example, multimedia streaming) closer to the users and speed up its delivery. The CDNs

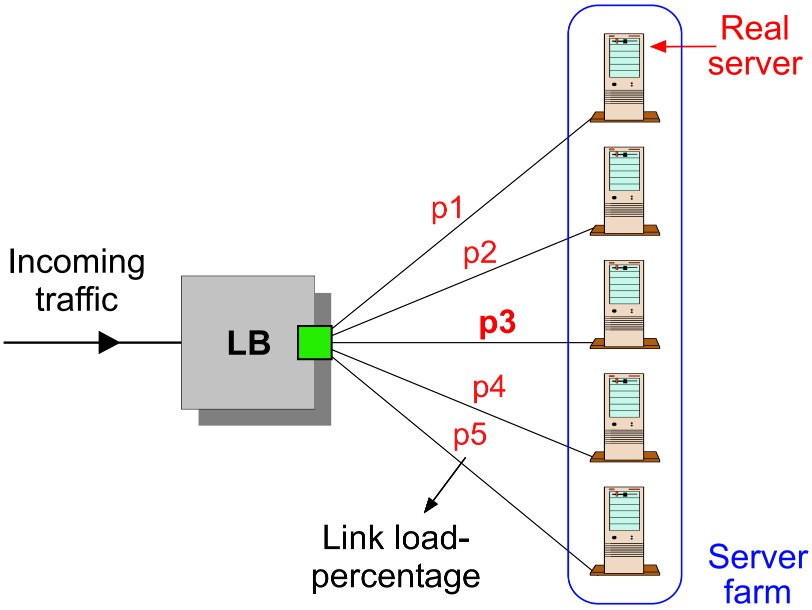


Figure 2.5: Server load balancing topology.

are an evolution of the caches. When the Web user clicks on a URL, the content-delivery network re-routes that user's request away from the site's original server to a cache server closer to the user. The three main techniques for the redirection are HTTP redirection, Internet Protocol (IP) redirection, and domain name system (DNS) redirection. In general, DNS redirection to the cache server is the most effective technique. The cache server determines what part of the content of the request exists in the cache, serves that content, and retrieves any non-cached content from the originating server. Any new content is also cached locally. Other than faster loading times, the process is generally transparent to the user, except that the URL served may be different from the one requested. CDN technologies are developed for example by Akamai [56] or Speedera [57].

2.4.3 Server farm load balancing

This is the most common technique of scaling a Web server capacity. As shown in Fig. 2.5 a pool of web servers, forming a server farm together, is connected to the Internet by a load balancer that acts as a front-end machine that intelligently directs the clients (their TCP connections) to the servers according to the servers' capacities and status. This allows each server to operate more efficiently. The choice of the server falls into two basic categories: content unaware and content aware.

In the *content unaware* case there are no dedicated web servers for some specific protocols and thus all of them can process any kind of request. Every time the load balancer receives a request for a connection (SYN), it redirects the TCP connection to a server that will execute the TCP three-way handshake. The redirection is generally based on Layer 2, 3 or 4 information such as IP addresses or ports numbers.

In the *content aware* case, the load balancer is an end-point of the TCP connection, and can interpret the user's request through parsing the layer-7 information (i.e. the URL). The redirection happens after the TCP three-way handshake between the load balancer and the user. Each server may contain different contents or possess different levels of security. This solution is more flexible, but more demanding than the unaware case, considering that the load balancer must be able to parse layer-7 information spread over several packets.

It is typically required that the load-balancing solution is able to ensure *sticky connections*, that is, connections that are always redirected to the same server. Some applications require sticky connections, like, for example, filling of forms, shopping carts or bank transactions.

These particular connections are typically handled in three ways:

- The load balancer stores the pair [user ID, IP address] into a table. This solution does not work with the Network Address Translation (NAT), because for each new connection of the same user, there may be a new, dynamically assigned source IP address;

- The real server or the load balancer are able to distinguish a sticky-connection and send a cookie back to the user that is used next time to redirect the connection. This solution does not work with encrypted connections;

- The real server or the load balancer uses HTTP to redirect to the correct real server.

DNS-based server load balancing

A readily available software load balancer is the Berkeley Internet Name Daemon (BIND) [12], developed by the Internet Software Consortium (ISC). It resides in the primary web server and intercepts packets as they enter the web site. When a DNS request arrives, it uses the DNS Round Robin to select a particular IP address from a pool of addresses. The selection pointer selects the addresses in a round-robin fashion. This solution has numerous drawbacks:

- Even if the load balancer spreads the connections evenly, there is no relationship to the load of individual connections. Thus some servers might have to handle much more load than others.
- If the primary web server fails, the entire farm is disabled.
- No consideration about the type of content requested.
- Does not work well if session state must be maintained.
- Does not work well if the servers are of heterogenous capacity.
- With large server farms, DNS Round Robin is difficult to configure.

Shaikh, Tewari and Agrawal [54] conduct a study on the effectiveness of a DNS-based server selection. The authors identify as a significant drawback the fact that DNS-based schemes typically disable client-side caching of name resolution results, thus increasing the resolution overhead by up to two orders of magnitude.

Advanced load balancers

A large variety of more complex load balancers exist nowadays. We provide a brief overview of some of the commercially available products.

The Central Dispatch from Resonate [48] is a software-based server management solution. It employs port load balancing: after creating the additional server processes in the actual server cluster, Central Dispatch users simply specify the range of ports on the server to which incoming requests should be mapped. The incoming requests are then forwarded to the most suitable server based on predefined scheduling policies. When port load balancing is configured, port selection at the server is performed, for example, in a round-robin manner; if one port does not respond, another port on the same machine is selected.

The Windows NT Load Balancer Service (WLBS) [41] from Microsoft is controlled by a *distributed management* software, which distributes the incoming load of IP requests across a cluster of multiple Windows nodes. The load-balancing scheme is based on the robust hash routing algorithm, described in Section 2.2.3, that incorporates the client IP address, its port number, or both, to determine which server responds. It is possible to specify a load percentage for each server. When changes occur, the load balancer starts a convergence process that automatically reconciles the changes in the cluster and transparently redistributes the incoming load.

IBM Network Dispatcher [24] is a software tool that routes TCP connections to multiple servers that share their workload, based on a monitored load metric. The algorithm contains an adaptive control loop, but it is required to *maintain state information* where each TCP connection has been mapped.

A *dedicated load balancer* is a configurable stand-alone appliance, which offers some router-like functionalities or is tightly integrated with an existing router system. Examples of such load balancer appliances are the Equalizer from Coyote Point Systems [60], BIG-IP LoadBalancer from F5 Networks and Cisco's Local Director.

The F5 Networks BIG-IP LoadBalancer 520 [22], is situated between the network and the server farm and automatically routes incoming queries to the most available server. It also provides support for heterogeneous server farms. The load balancer intercepts all data packets addressed to the site's IP address and distributes them to the appropriate server. The supported load-balancing algorithms are:

- *Fastest*–user connections are passed to the available server that responds the fastest;

- *Round Robin*–traffic is sent to the next available server in a predetermined sequence;

- *Least Connections*–the user is connected to the server with the least number of current connections;
- *Ratio*–assigns fractions to the server that best fits the request, according to a system that assigns weights for various factors, such as server capacity.

Local Director from Cisco Systems [59] is a connection manager appliance, tightly integrated with the Cisco routing and switching ("forwarding agent") products. When a forwarding agent receives a connection request, the request is forwarded to the manager (Local Director). The manager makes the load balancing decision and instructs the forwarding agent with the optimal destination. After destination selection, session data is forwarded directly to the destination without further manager participation. The load-balancing decisions can be based on various algorithms: application availability, server capacity, round robin, least connections, or Dynamic Feedback Protocol (DFP).

Another family of server load-balancing devices falls into the category of *content smart switches and routers*. These are standard switches or routers with some extra features enabled, including server load balancing. Examples are the suite of Alteon switches from Nortel Networks [42] or the Content Smart Switches and Routers from Cisco Systems [58]. Typically, they support load balancing over various entities, like servers, firewalls or multi-homed links. The server selection is based on server load and application response time, or the least connections or the round-robin algorithms.

2.5 Conclusions

In this chapter, we have reviewed the previous research and developments in the areas relevant to this thesis—in the load-sharing field and in the field of router architecture and of web server farm load balancing, as examples of multiprocessor systems within networks. In the load-sharing discussion, we have covered in depth the highest random weight (HRW) mapping algorithm, as it is a foundation for our adaptive load-sharing method presented in Chapter 4.

Some conclusions with respect to the topic of this thesis can be reached based on the works reviewed. Firstly, it is a clear trend, primarily for the reasons of scalability and fault tolerance, to equip network nodes with multiple processors. However, there are open questions as to what kind of architecture exploits such a multiprocessor system best.

Other findings show that it is non-trivial to design a hash-based, and thus fast, load-sharing method that would yield reasonable load balancing performance and yet not be computationally intensive. In the field of web servers, clearly the load-sharing techniques rely primarily on maintaining some form of state information about the connection mapping, regardless of the discipline used for the mapping establishment. This results in good load balancing performance, as the discipline can be periodically adapted, yet at a high cost of maintaining large amount of state information. It is the goal of this thesis to develop a method that preserves the advantages and overcomes the drawbacks of the techniques discussed.

Chapter 3

Optimizing router architecture

3.1 Introduction

In this preparatory chapter, a model of an essentially distributed router is optimized from various perspectives, to obtain a more detailed understanding of the issues in the router architecture design space.

In the first part, we optimize the model from the perspective of the total cost of the entire system. The objective of the method is to serve as a general means for optimizing a router architecture with a given set of input constraints. The constraints are maximum line interface bandwidth, number of router line cards and maximum packet processing delay within the system.

To carry out the optimization on a realistic system model, the system model contains further constraints, which can be interpreted as technological limits, such as the maximum processing power of the forwarding engines (FEs) or the maximum switch port speed. The full set of constraints defines a space of feasible solutions over which the optimization is carried out. The optimization cost function is an aggregate of estimated market-based costs of the individual elements. The cost is expressed as a function of the technological parameters of each element.

The optimization output consists of variables describing the optimal architecture– the processing power of the local forwarding engines (LFEs) and of the main forwarding engines (MFEs), number of MFEs, switch port speed, and the distribution of packet processing among LFEs and MFEs. Based on the results of the optimization, we reach some general conclusions about the most economical router architecture for a given set of constraints.

In the final section of this chapter, optimization in terms of maximizing the processing capacity while remaining within certain performance constraints is discussed. We define the term of acceptable load sharing and show how abiding by such principle routers may increase their processing capacity. Finally, some conclusions with respect to the applicability of load sharing within a router system are presented.

The chapter is organized as follows: Section 3.2 presents the router architecture model, and, in Section 3.3, the cost optimization problem is described in detail. In Section 3.4, results of the most interesting cost optimizations are discussed; Section 3.5 explores router optimization from the load-sharing point-of-view, and, finally, Section 3.6 contains some concluding remarks.

3.2 General distributed router architecture

3.2.1 Router model

A general model of an essentially distributed router architecture is optimized. Our model (see Fig. 3.1) contains a fixed number of LFEs (one per router input) of variable processing power (including null processing power, meaning that the LFEs are absent), a variable number of MFEs with variable processing power, and a switch of variable port speed. This work extends the model presented in [10] by a switch element and an LFE queuing and queue overload model. Furthermore, we present a hybrid, more general router architecture model, encompassing a large space of possible, in essence distributed, router architectures, including the centralized, fully distributed and parallel cases presented in [10].

We consider a router having k LCs, with an LFE attached at every LC (see Fig. 3.1). A switch element interconnects all the LFEs and the pool of m parallel MFEs. All the possible sequences the processing of a packet may take–at an LFE, at an MFE, or at both processing units—are accounted for. A fraction $r \in [0, 1]$ of the incoming

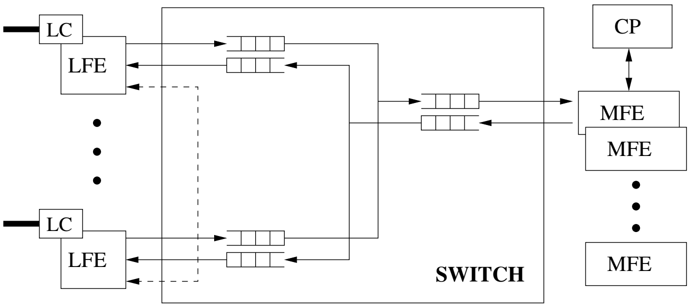


Figure 3.1: LFEs, switch and MFEs within the general distributed router architecture model.

traffic is processed locally at the LFEs; the fraction $1 - r$ is diverted directly to the MFEs without being enqueued at the LFEs (see Fig. 3.2). Thus, if $r = 0$, LFEs are not used at all and the router consists only of a pool of parallel MFEs and a switch. When $r > 1$ the LFE may not be able to handle all the traffic destined for it locally, for various reasons, such as for being overloaded (see Section 3.2.2). Such traffic is sent to the MFEs as well, but only after passing through the LFE. Thus, if $r = 1$, all the traffic is enqueued at the LFEs, yet a fraction that the LFEs will not be able to process will still subsequently be sent to the MFEs.

Arriving traffic is modelled as a Poisson process. The mean arrival rate at each input LC is λ_i packets per second (pps). The total router load is thus $\lambda = k\,\lambda_i$. The analysis carried out is a worst-case scenario, where we consider all the links to be fully loaded. In reality, workloads on different LCs are generally not uniform and may vary significantly over time. This implies that LFEs with a higher workload would forward more packets to the MFEs for processing than LFEs with a smaller workload, and one can imagine a feasible problem solution where for some periods of time, individual LFEs would be overloaded. We have experimented with nonuniform workload distributions on different LCs and the LFE overload, but the optimization results did not differ significantly from the uniform model. Nonuniform LC workloads are therefore not included in the model. The LFE model, as presented in Section 3.2.2, is applicable for the overload modelling, however the optimization never finds an overloaded LFE solution to be the optimal one.

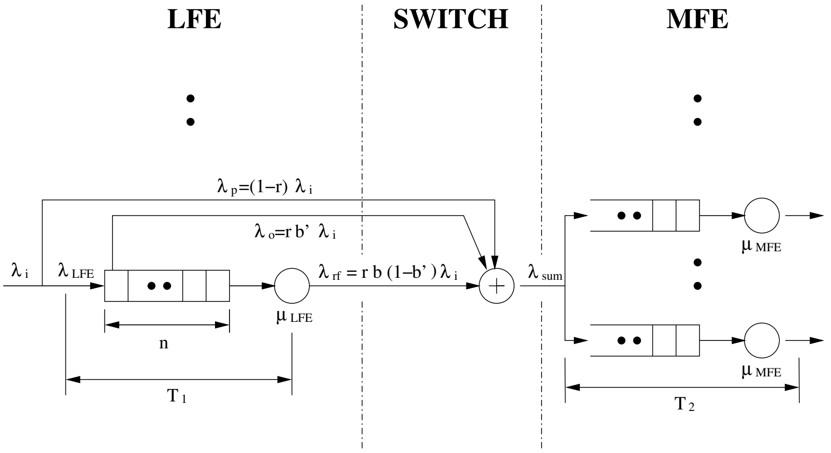


Figure 3.2: Model of the general distributed router architecture with multiple LFEs and MFEs.

With respect to the optimization, parameters k and λ_i are a part of the optimization input, whereas values of r and m are a part of the output.

3.2.2 LFE and MFE model

The processing power of an MFE is μ_{MFE} pps, and that of an LFE is μ_{LFE} pps. We use $\mu_{\max}$ as a bound on the maximum number of packets an FE can handle per second.

The possible scenarios of packet-processing distribution among LFEs and MFEs are depicted in Fig. 3.2. The fraction of traffic arriving at an LFE is $\lambda_{\mathrm{LFE}} = r\lambda_i$. A fraction $\lambda_p = (1-r)\lambda_i$ is pre-scheduled directly for processing at the MFEs.

Regarding the packets sent for resolution through the switch to the MFEs, we assume that it is only the packet control information, i.e. the packet header, that travels through the switch (as in [27]). The packet payload is assumed to be buffered until a resolution of the packet processing task arrives from the MFE pool, again, travelling through the switch. Thus, in terms of number of packets, the amount travelling through the switch is the same, yet in terms of bits, only a fraction of the packet size makes the trip to the MFEs. In this work, the processing overhead and the memory size requirements for the packet header detachment, the payload buffering, and the packet reassembly are not considered.

Furthermore, we assume that a single LFE workload, λ_{LFE}, can be greater than its processing power μ_{LFE}. The LFE queue size n is introduced as a parameter to model the LFE overload. When the LFE queue is full, a packet *cannot* be processed by the LFE and is forwarded to the MFE pool. Note that the overload traffic does not have a Poisson distribution because the probability that an LFE queue is full depends on the LFE load, but, for the sake of simplicity, we approximate it with a Poisson distribution as follows: the LFE queue is an $M/M/1/n$ queue. Thus, the probability of a packet arriving at a full LFE queue is (see [31]):

$$b'(\lambda_{\text{LFE}}) \quad = \quad P_n = \frac{1-\rho}{1-\rho^{n+1}}\rho^n, \qquad \rho = \lambda_{\text{LFE}}/\mu_{\text{LFE}}. \tag{3.1}$$

Thus, we assume the fraction of traffic $\lambda_o = b'\,\lambda_{\text{LFE}} = r\,b'\,\lambda_i$ to be sent to the MFE pool due to LFE overload.

Furthermore, as in [10], even if a packet *is* being processed by an LFE, with a fixed probability b the LFE will not be able to find the packet next hop, and the packet is likewise forwarded to the MFE pool. Such packets account for route table misses, for example when the LFE acts only as a cache, storing a fraction of the routing table. We denote such a fraction of traffic as $\lambda_{rf} = r\,b\,(1-b')\lambda_i$.

Finally, the fraction of traffic that actually does get resolved at the LFE and is forwarded directly to the outgoing switch port is $\lambda_q = \lambda_i - (\lambda_p + \lambda_o + \lambda_{rf}) = r(1-b)(1-b')\lambda_i$.

In Fig. 3.2 we observe that there are three possibilities for a packet to be queued. Either a packet is queued at an LFE and waits for time T_1, it is forwarded to the MFE and waits for T_2, or it is queued at both the LFE and the MFE owing to the LFE resolution failure.

Note that given the various paths the packet processing in the router can take, packets belonging to a particular flow may be reordered, which is highly undesirable [34]. In the interest of simplicity, we do not consider the additional processing overhead required to prevent reordering in this section.

LFE processing time

The average number of packets in a processor, the average workload arriving at the LFE queue, and the average LFE response time are

$$\begin{aligned}
\overline{N}(\lambda_{\text{LFE}}) &= \rho \times \frac{n\rho^{n+1} - (n+1)\rho^n + 1}{\rho^{n+2} - \rho^{n+1} - \rho + 1} && (3.2) \\
\lambda_a &= \lambda_{\text{LFE}} * (1 - P_n) = \lambda_{\text{LFE}} * \frac{1-\rho^n}{1-\rho^{n+1}} && (3.3) \\
\overline{W}(\lambda_{\text{LFE}}) &= \frac{\overline{N}(\lambda_{\text{LFE}})}{\lambda_a} = \frac{1}{\mu} \times \frac{1-\rho^{n+1}}{1-\rho^n} \times \frac{n\rho^{n+1} - (n+1)\rho^n + 1}{\rho^{n+2} - \rho^{n+1} - \rho + 1}\ . && (3.4)
\end{aligned}$$

Observing the behavior of a saturated LFE, we see from Eq. (3.4) that for higher n, the waiting time is longer. Therefore, a router with long LFE queues would be penalized with respect to the packet delay time compared to an equivalent router with smaller queues. On the other hand, a router with extremely small LFE queues (e.g. $n = 1$) would have frequent queue overflows even on the nonsaturated LFEs, which would again penalize its performance. The queue length n thus has to be chosen carefully in order to achieve optimal performance. Ideally, n should be included in the optimization of the system parameters. Experimentally, though, we have found that changes in n do not have a very significant influence on the optimization in comparison to other factors, especially as the optimization never finds an overloaded LFE to be the optimal solution. Thus, to simplify the analysis, we use fixed n values. Note however that in reality, a queue of larger size would be necessary to handle bursty traffic, for which we do not account for in our model. Time T_1 is simply the average response time $\overline{W}(\lambda_{\text{LFE}})$.

MFE pool processing time

The MFE pool queue is, as in [10], a simple infinite $M/M/m$ queue, with the input workload representing the sum of non-processed packets from the LFEs, together with the pre-scheduled packets. As the part of workload sent for resolution to the MFE represents a sum of Poisson processes, the sum is a Poisson process as well. This workload and the corresponding $M/M/m$ queue waiting time are on average [31]:

$$\begin{aligned}\lambda_{\text{sum}} &= k\,(\lambda_p + \lambda_o + \lambda_{rf}) \quad (3.5)\\ T_2 &= \frac{1}{\mu_{\text{MFE}}} + \frac{\rho P_Q}{\lambda_{\text{sum}}(1-\rho)}, \quad (3.6)\end{aligned}$$

where

$$\rho = \frac{\lambda_{\text{sum}}}{m\mu_{\text{MFE}}}\ , \quad P_0 = \frac{1}{\sum\limits_{j=0}^{m-1} \frac{(m\rho)^j}{j!} + \frac{(m\rho)^m}{m!\,(1-\rho)}}\ , \quad P_Q = \frac{P_0 (m\rho)^m}{m!\,(1-\rho)}\ . \quad (3.7)$$

Considering the MFE pool an $M/M/m$ queue may not necessarily be realistic, as we again do not account for the overhead to prevent packet reordering. Either a load balancing device in front of the pool, or packet re-sequencer at the exit of the pool would have to be in charge of maintaining the packet order. However, such mechanism may prevent the pool from acting as an $M/M/m$ queue. For simplicity, we do not consider the devices and overhead required to ensure the packet sequence.

3.2.3 Switch model

General input/output switch

The switch is characterized by two parameters—the switch port speed s and the number of input ports k. As k is an input to the optimization, s is the only parameter optimized by the method. The switch port speed is expressed in terms of transmission time per the fixed size switch cell, that is, in seconds per cell. The parameter s is constrained from below by the technological limit of s_{min}, $s > s_{\text{min}}$, which means that a fixed-size switch cell cannot be transmitted at a switch port in less than s_{min} seconds. To avoid confusion with the intensity indicators in packets per second, the intensity indicators in switch cells per second are denoted with *, e.g. λ^* instead of λ.

We include the switch in our model by introducing the switch delay. In order to model the switch delay time, we consider a formula for an input/output-queued switch derived in [26] and [25]. In the interest of simplicity, we assume that the switch has infinitely large output queues, and that the number of inputs is large (i.e. greater than 16, [26]). In the case of a lower number of inputs, the performance of the switch is actually better than the formula depicts (as described in [30]), owing to lower contention, but in such a case the formula can still be used as a rough upper bound. Note that the head-of-line congestion at the input port considered in [26] and [25] has been eliminated in the latest switch architectures; however, in this work, we conform to this model in order to obtain a simple analytical formula for computing the switch delay.

A cell arriving at an input port first waits at the input queue, then at the head of the input queue because of head-of-line congestion, and, finally, at the switch output port (see Fig. 3.1). We denote $W_i(\lambda_x^*)$ as the average waiting time until the head of the input line is reached. The term λ_x^* denotes the intensity of the input traffic (in switch cells per second). This is an $M/G/1$ queue with service time $W_b(\lambda_x^*)$ equal to the time a cell spends waiting at the head of the input queue owing to head-of-line congestion. We denote $D_{\text{out}}(\lambda_y^*)$ as the average delay from the instant a cell appears at the head of its input queue until the instant it begins transmission at the output port. As shown in [26], this is an $M/D/1$ queuing system, as we assume the switching matrix speed to be constant and the input traffic to be Poisson. The term λ_y^* is the intensity of the aggregated Poisson flow arriving at the output port (in switch cells per second).

The switch traffic consists of k equally loaded *local ports* and one additional port, the *master port*, used for transferring the packet headers to and from the MFE pool (see Fig. 3.3). As we consider infinite output queue sizes, waiting time due to head-of-line congestion is $W_b(\lambda_x^*) = 0$ because a cell can be queued at the output the moment it arrives at the head of an input queue. It follows from [26, Eq. (2.9)] that

$$W_i(\lambda_x^*) = \frac{\lambda_x^* s^2}{2(1 - \lambda_x^* s)},$$

where λ_x^* denotes the intensity of the input traffic (in switch cells per second) and s denotes the switching matrix processing time. From the analysis of an $M/D/1$ queue

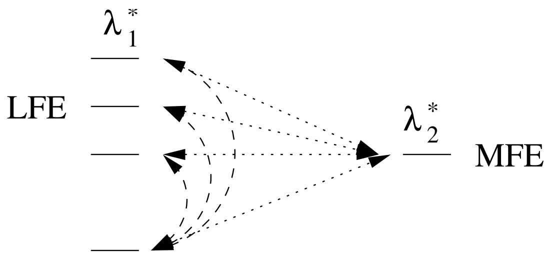


Figure 3.3: Traffic flows within the switch; λ_1^* represents the amount of traffic leaving the local port (equal to the amount arriving at the local port), and λ_2^* represents the amount of traffic arriving at the master port (equal to the amount that leaves the master port).

we have

$$D_{\text{out}}(\lambda_y^*) = \frac{\lambda_y^* s^2}{2(1-\lambda_y^* s)},$$

where λ_y^* is the intensity of the aggregated Poisson flow arriving at the output port (in switch cells per second).

In order to exploit the above switch model in conjunction with the FE models, we need to transform the load variables to a common unit: packets per second. As Internet packets are *variably sized*, we need to use some approximations to establish a relationship to the *fixed-size* switch cells. Two kinds of packets travel through the switch—entire packets, and the packet headers traveling between the LFEs and the MFE pool (see Fig. 3.3). Let $\bar{S}_P$ denote the average size of a packet in the router incoming traffic, and $\bar{S}_H$ the average size of a header message traveling in the switch. We assume that a packet header corresponds in size to a single switch cell. We denote c as the ratio between the packet header cell size and the average packet size, $c = \bar{S}_H/\bar{S}_P$. Thus an average packet accounts for $1/c$ switch cells. Measurements and analysis have shown that the Internet traffic distribution is highly nonuniform. Empirical values in recent studies show $\bar{S}_P \doteq 300$ B [63]. A typical switch cell size is 64 B, with the payload part being equal to 60 B. Thus, a 40–44 B packet header usually fits into a single such cell, $\bar{S}_H = 64$ B and, consequently, $c \doteq 0.2$ is a good typical value.

The transfer of packet headers among the processing engines may be time consuming, especially when the switch traffic is already high. In the interest of simplicity, we assume that the two kinds of traffic travelling within the switch, entire packets and packet headers, are not distinguished in any way by the switch element. Thus, the packet header communication overhead traffic saturates the switch further. Note that with the latest switch designs, full decoupling of the two kinds of traffic may be achievable by using different switch priorities or separate switch planes. The following paragraphs describe the introduction of the overhead into the model.

A fraction of traffic is sent to the MFE pool for processing. With reference to Fig. 3.3, we have

$$\lambda_1^* = \frac{\lambda_i}{c} + (\lambda_p + \lambda_o + \lambda_{rf}), \qquad \lambda_2^* = k\,(\lambda_p + \lambda_o + \lambda_{rf}), \tag{3.8}$$

where λ_1^* is the total switch outgoing traffic at an LFE port (comprising both packet headers and entire packets) and λ_2^* is the switch outgoing traffic at the master port (comprising packet headers only). The values of λ_1^* and λ_2^* are expressed in switch cells per second, whereas $\lambda_i, \lambda_p, \lambda_o$, and λ_{rf} are expressed in packets per second.

From the above discussion, we see that the waiting time for a packet header traveling to the MFEs is $W_i(\lambda_1^*) + D_{\text{out}}(\lambda_2^*)$ and that the waiting time for the way back to the output port is $W_i(\lambda_2^*) + D_{\text{out}}(\lambda_1^*)$, hence the total switch delay for a packet header trip is

$$D_H(\lambda_i) = \frac{\lambda_1^* s^2}{1 - \lambda_1^* s} + \frac{\lambda_2^* s^2}{1 - \lambda_2^* s}. \tag{3.9}$$

An entire packet traveling through the switch after its destination output port has been found by the next-hop resolution process experiences a per-cell delay of $W_i(\lambda_1^*)$ at the switch input and a per-cell delay of $D_{\text{out}}(\lambda_1^*)$ at the output port, and occupies on average $1/c$ switch cells. Thus the mean switch delay for the entire packet is

$$D_P\,(\lambda_i) = \frac{1}{c}\,(W_i(\lambda_1^*) + D_{\text{out}}(\lambda_1^*)) = \frac{\lambda_1^* s^2}{c\,(1 - \lambda_1^* s)}. \tag{3.10}$$

3.2.4 Time a packet spends within the system

The mean time T a packet spends in the system is

$$\begin{aligned} T \;=\; & \frac{\lambda_i - \lambda_p - \lambda_o}{\lambda_i}\, T_1 \;+\; \frac{\lambda_p + \lambda_o + \lambda_{rf}}{\lambda_i}\, T_2 \;+ \\ & +\; \frac{\lambda_p + \lambda_o + \lambda_{rf}}{\lambda_i}\, D_H(\lambda_i) \;+\; D_P(\lambda_i), \end{aligned} \tag{3.11}$$

where the first element represents the fraction of packets processed by the LFEs, the second the fraction of packets processed by the MFEs, the third the switch delay for a remotely processed packet header, and the fourth the switch delay for the entire packet traveling through the switch.

3.3 Cost optimization

3.3.1 Forwarding engine cost

To simplify the problem, the cost associated with an FE is assumed to be a linear function of processing power of the form:

$$\text{cost}_{\text{MFE}}\ (\text{US }\$) \doteq c_1\ \mu_{\text{MFE}}, \quad \text{cost}_{\text{LFE}}\ (\text{US }\$) \doteq c_2\ \mu_{\text{LFE}}, \tag{3.12}$$

where μ_{LFE} and μ_{MFE} are expressed in packets per second. We denote a as the ratio of the two coefficients, $a = c_1/c_2$.

3.3.2 Switch cost

Effective switch throughput

Recall that s is the switch speed, and k the number of input ports. In order to establish a general measure of the switch performance, we define the *saturation throughput* of the switch to be $\lambda_s^* = k/s$ (in switch cells per second). The notion comes from the fact that the switch delay,

$$D(\lambda_i^*) = \frac{\lambda_i^* \, s^2}{(1 - \lambda_i^* \, s)} \,, \tag{3.13}$$

tends to infinity as the per-input switch load λ_i^* (in switch cells per second) approaches $1/s$ (see [26]).

We denote as the *effective switch throughput* $\lambda_e^* = \alpha \lambda_s^*$ a fraction α of the saturation throughput for which the delay remains within reasonable bounds. For Internet applications, we assume $\alpha = 0.9$. A common switch performance metric used by the industry is the switch effective throughput λ_e' measured in bps. The relationship between λ_e^* and λ_e' holds as:

$$\lambda_e' = 8 \, \bar{S}_H \, \lambda_e^* \,. \tag{3.14}$$

Switch cost function

To establish a switch cost function $\mathrm{cost}_S(s, k)$ dependent on the switch performance, we establish a relationship between λ_e' in bps and the switch cost in US \$. We assume that within certain limits of $k \leq k_0$, there is a linear dependency of the switch cost on the switch effective throughput λ_e', which can be characterized by the following formula:

$$\mathrm{cost}_S(s, k) \text{ (US \$)} \doteq c_3 \, \lambda_e'(s, k), \text{ where } k \leq k_0. \tag{3.15}$$

The limit of k_0 denotes the limit for single-stage switches. For $k > k_0$, we assume more aggressive growth of the switch cost with λ_e', because we assume that such a switch can only be built using a multistage architecture. A typical empirical value of k_0, which is also used in this work, is $k_0 = 64$. From the fixed point of k_0, we assume a cost function dependency of linear-logarithmic form, approximately:

$$\mathrm{cost}_S(s, k) \text{ (US \$)} \doteq c_3 \, \lambda_e'(s, k) \, \log(\lambda_e'(s, k)), \text{ where } k > k_0. \tag{3.16}$$

In the interest of a smooth transition and reasonable values, we normalize the function as followings:

$$\mathrm{cost}_S(s,k)\ (\text{US } \$) \doteq c_3\, \lambda'_e(s,k)\, \log\left(e + \frac{\lambda'_e(s,k) - \lambda'_e(s,k_0)}{\lambda'_e(s_{\min},k_0)}\right), \text{ where } k > k_0. \tag{3.17}$$

3.3.3 Total system cost

The total system cost formula holds as

$$\mathrm{cost} = m\, c_1\, \mu_{\mathrm{MFE}} + k\, c_2\, \mu_{\mathrm{LFE}} + \mathrm{cost}_S(s,k). \tag{3.18}$$

3.3.4 Optimization problem

The cost is optimized over the tunable system parameters: $(r, \mu_{\mathrm{MFE}}, \mu_{\mathrm{LFE}}, s, m)$. Given the maximum allowed mean packet processing time $T_{\max}$, we derive the following *optimization problem*:

Optimization function	Parameters	Constraints
$m\, c_1\, \mu_{\mathrm{MFE}} + k\, c_2\, \mu_{\mathrm{LFE}} + cost_S(s,k)$	$\{r, \mu_{\mathrm{MFE}}, \mu_{\mathrm{LFE}}, s, m\}$	$0 \leq r \leq 1$
		$0 \leq \mu_{\mathrm{MFE}} < \mu_{\max}$
		$0 \leq \mu_{\mathrm{LFE}} < \mu_{\max}$
		$s_{\min} < s$
		$T < T_{\max}$
		$1 \leq m.$

Note that m is an integer, whereas all the other optimized parameters are rational numbers.

3.4 Numerical results

3.4.1 Optimization

Numerical results have been obtained using the Matlab Optimization Toolbox environment. First, the value of m is increased until there exists a feasible solution, and then the constrained nonlinear optimization function `fmincon` is used to find the optimum. The system variables in our optimizations have been set to the following values: $T_{\max} = 10^{-6}$ s, $c_2 = 6 \times 10^{-5}$, $c_3 = 10^{-8}$, $\mu_{\max} = 6 \times 10^6$ pps, $s_{\min} = 10^{-9}$ s (meaning a fixed-size switch cell would have to be transmitted at a switch port within 1 ns), $n = 100$. The values have been selected to approximate the limited market information available.

The ratio of costs of equally powerful LFEs and MFEs, a, is alternated in our simulations over values $a \in \{2, 10, 100\}$ and influences several variables. Variable c_1 is dependent on c_2 and a, $c_1 = ac_2$. The value of b reflects a in the following manner: we assume that a can be interpreted as a difference in memory size available to the processing engines. Thus a determines a fraction of the memory available at an LFE, and b, the fraction of nonresolvable packets at an LFE, reflects the cache miss rate at an LFE. A sample dependency between a cache hit-rate and cache size can be found for example in [40]. Based on that, we use the following pairs of (a, b): (2, 0.1), (10, 0.2), (100, 0.9).

Figures 3.4 - 3.16 show plots of the output variables over a spectrum of the number of inputs k and link speeds λ_i. The number of inputs k grows geometrically with a coefficient of 2, $k \in \{8, 16, 32, ..., 1024\}$. The values on the maximum interface bandwidth λ_i axis grow geometrically with a coefficient of $\sqrt{2}$, thus including link speeds approximately corresponding to the capacities of 10 Mb – 1 Gb Ethernet and OC-48 – OC-768 links, in Mpps, that is, $\lambda_i \in \{0.025, 0.035, 0.050, ..., 6.40, 9.05\}$ Mpps.

3.4.2 Total system cost

Figure 3.4 depicts the system cost on a linear and a logarithmic plot for $a = 10$. The cost grows exponentially along both the k and λ_i dimensions, with a steeper increase in the high λ_i segment, which is due to the increasing influence of the switch cost on the total cost.

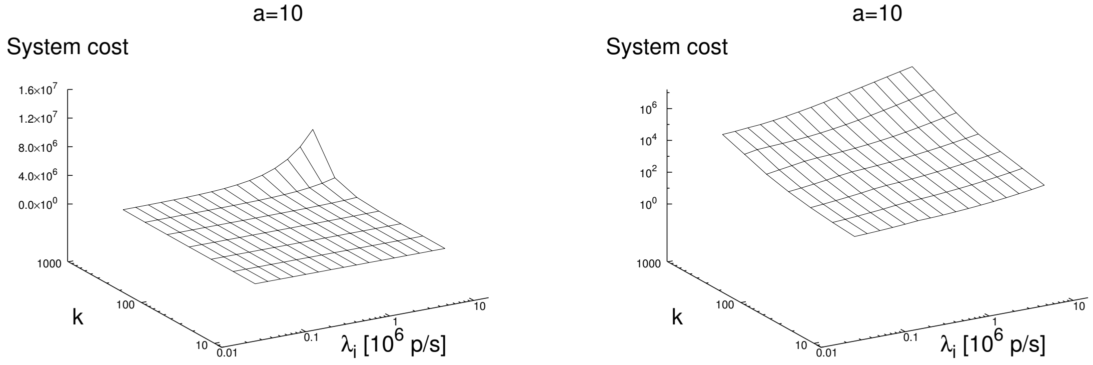


Figure 3.4: Total system cost.

3.4.3 Distribution of resources—LFEs, MFEs and switch capacity

Figure 3.5 shows how the optimization selects between the two extreme points of the solution space. For a smaller part of the problem space, r attains the value of 0, meaning no LFEs are needed. When r is equal to 1, the results indicate that it is cheaper to add and fully stress the LFEs to be able to handle the load.

It is clearly visible that a certain boundary divides the problem space into two regions. Typically, for systems with a small number of inputs and small link loads, deploying LFEs is too expensive and $r = 0$. This somewhat counter-intuitive result comes from the fact that the processing capacity needed for one $M/M/1$ queue serving the aggregate load is smaller than the processing capacity needed for k $M/M/1$ queues, each serving $1/k$ of the aggregate load, if both systems need to conform to the same time constraint. As the relationship is nonlinear, with the increase in the total amount of processing capacity required, at a certain boundary it becomes more economical to use the multiple, less powerful processors, and thus, for systems with a higher number of ports or higher link loads, LFEs are more economical to fulfill the performance requirements and $r = 1$. The exact curve of the shift differs for various a. Two further observations can be made. For $a = 2$ and $a = 100$, r starts to decrease from 1 towards 0 for very high λ_i. Furthermore, for $a = 100$ and very high

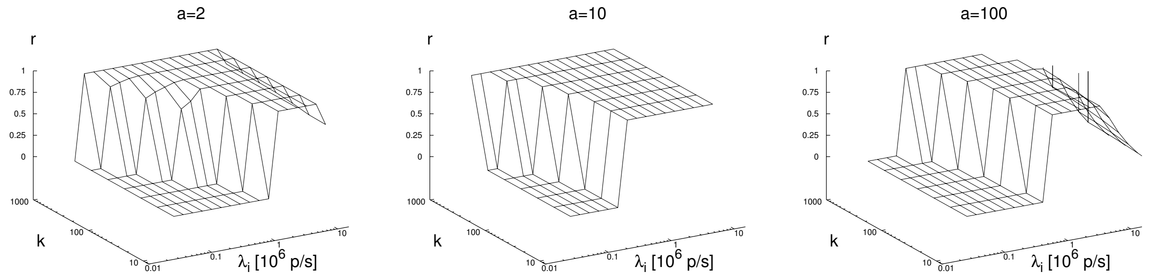


Figure 3.5: Fraction of traffic enqueued at LFEs, r.

k and λ_i, r is undefined, meaning that it is not feasible to build a router with the constraints given. Observing the optimization results of other parameters leads to a better understanding of these phenomena.

Figures 3.6 and 3.7 show on a logarithmic scale the optimal amount of total processing power within the system and its cost. Figures 3.8 and 3.9 show on a logarithmic scale the optimal switch, characterized by the saturation throughput, and its cost.

Figures 3.6 to 3.9 explain the differences in the shape of the boundary between $r = 0$ and $r = 1$ in Fig. 3.5. We observe a trade-off between increasing the switch throughput and deploying LFEs. The influence of the switch cost, however, differs when the factor a changes. The lower a, the higher the influence of the switch cost on the boundary shape, because when a and b are low, the total system cost remains lower and the switch cost influences the trade-off between $r = 0$ and $r = 1$, described above. However, when a is large, b becomes large as well, and a large fraction of the packets processed at the LFEs have to travel to the MFEs anyway, thus the introduction of LFEs is beneficial only for very high link speeds. The MFEs and the system in total are then very expensive, and the switch cost is no longer a factor in the trade-off.

3.4.4 Distribution of processing capacity—LFEs and MFEs

Figure 3.10 shows the optimal amount of processing power at each individual LFE. For $a = 2$ and $a = 100$, the LFE processing power μ_{LFE} reaches the upper limit $\mu_{\max}$ in

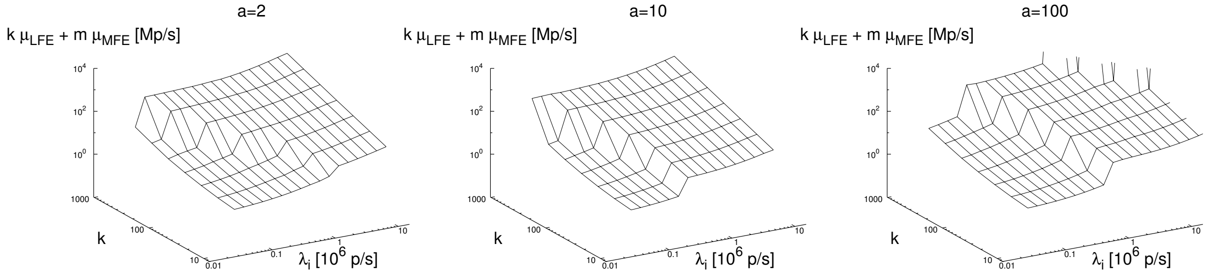


Figure 3.6: Total processing power of the FEs.

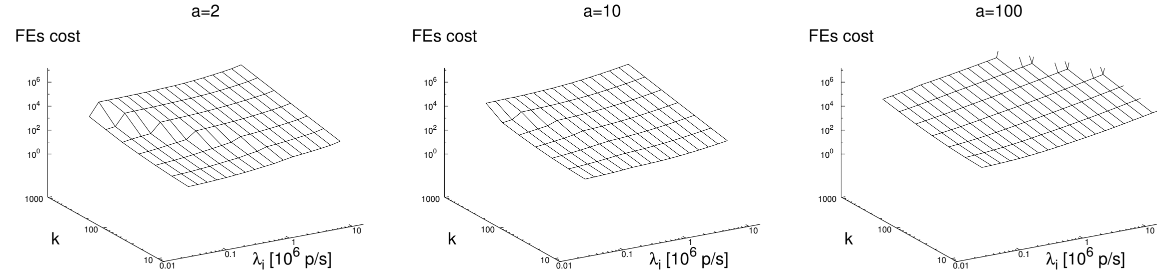


Figure 3.7: Total cost of the FEs.

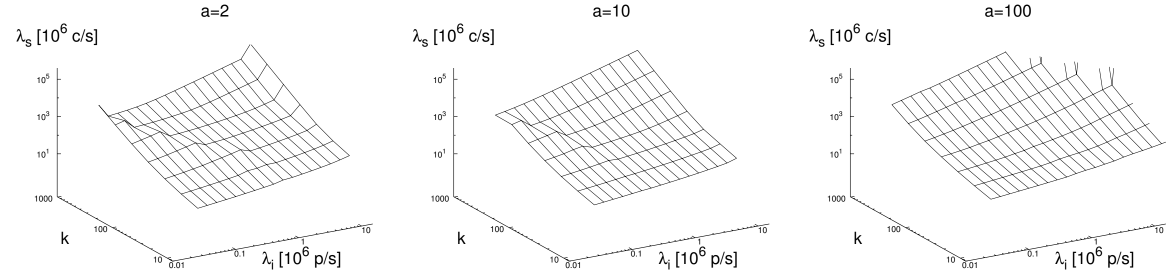


Figure 3.8: Switch saturation throughput.

the very high λ_i region. As the LFE is overloaded at that point, it does not make sense to enqueue additional packets at the LFEs because they only incur additional delay. Thus it is more efficient to send the appropriate fraction of the packet headers directly to the MFEs, resulting in a decrease of r. The boundary where the LFE processing power limit is reached differs for various a. This phenomenon is again dependent on the particular pairing of (a, b), which, as described in Section 3.4.3, influences the boundary where it becomes advantageous to use LFEs, and, in particular, on the

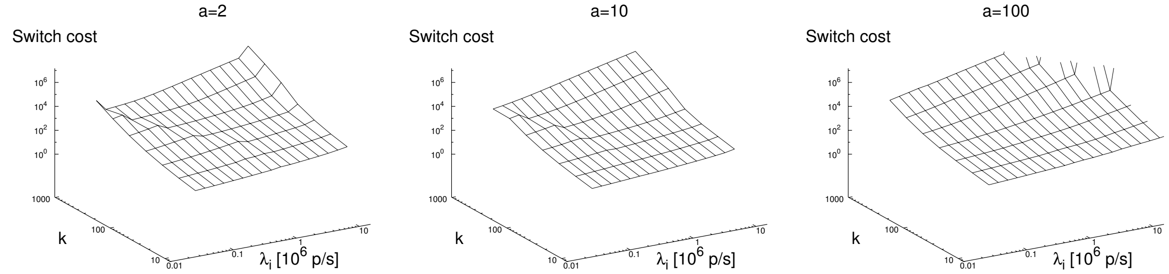


Figure 3.9: Switch cost.

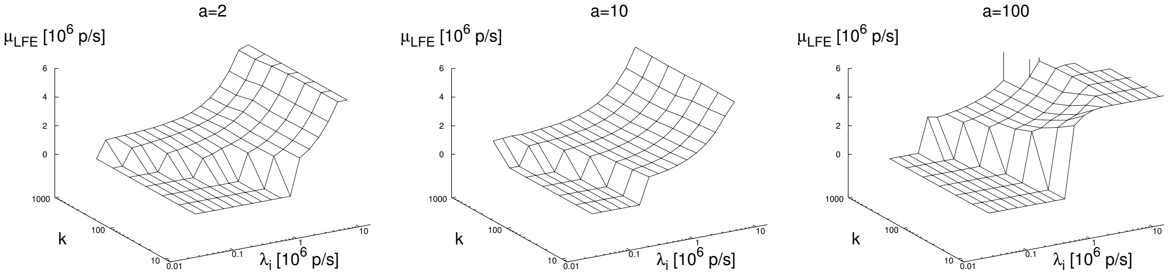


Figure 3.10: Optimal processing power per individual LFE.

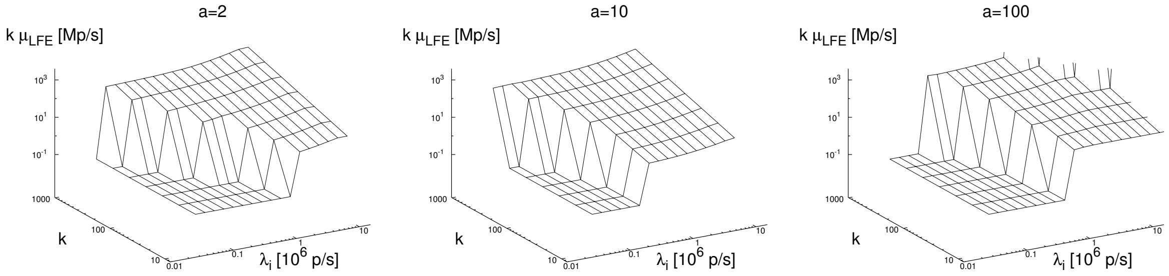


Figure 3.11: Total processing power of the LFEs (note that the graph was adjusted for the reader's convenience in order to be able to depict the values equal to 0, which would normally tend to negative infinity on a logarithmic graph).

fraction of the LFE resolution failures b, which increases the required LFE processing power.

Figures 3.11 and 3.12 show the total amount of processing capacity of the LFEs and the MFEs, respectively. Figure 3.13 shows how the total amount of processing power is partitioned among the LFEs and MFEs. In most of the problem space,

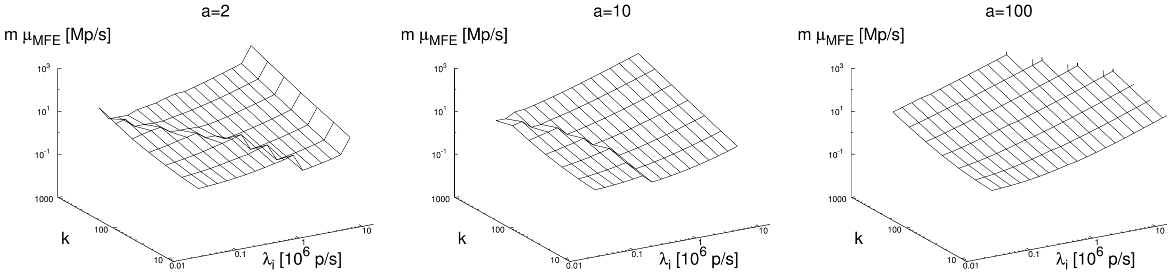


Figure 3.12: Total processing power of the MFEs.

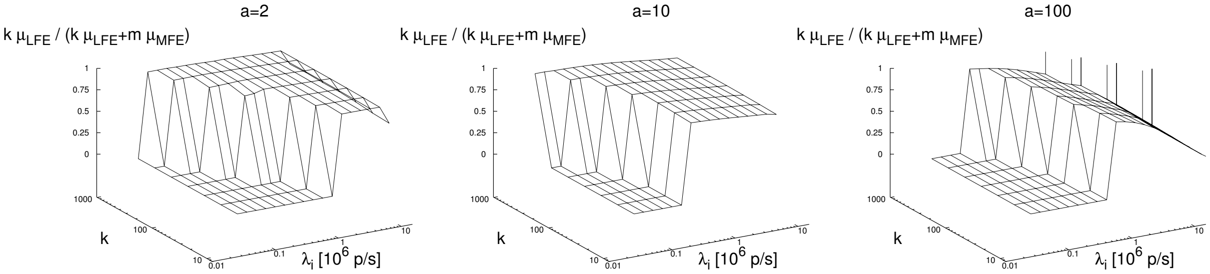


Figure 3.13: Fraction of LFE processing power out of total system processing power.

the major part of the optimal system processing power rests with the LFEs. In the segment of routers with high-speed links, for $a = 2$ and $a = 100$, the LFE processing power limit $\mu_{\max}$ is reached and thus the bulk of the processing capacity begins to shift towards the MFEs. At the same time, as described in Section 3.4.3, in the segment of devices with few, low-speed links, LFEs are not used at all and thus their share of the total capacity is equal to 0.

3.4.5 Switch speed

Figure 3.14, 3.15 and 3.16 depict the optimal switch port transmission speed and the overhead of the header-passing traffic compared to the total switch traffic. We see why there is no feasible solution for $a = 100$. Given the dependence of b on a, a large fraction of traffic ($b = 0.9$) fails to be resolved at the LFEs and still travels through the switch to the MFE pool. Thus, LFEs cannot be used to a great extent to decrease the packet delay in a router, and the switch, the master port in particular,

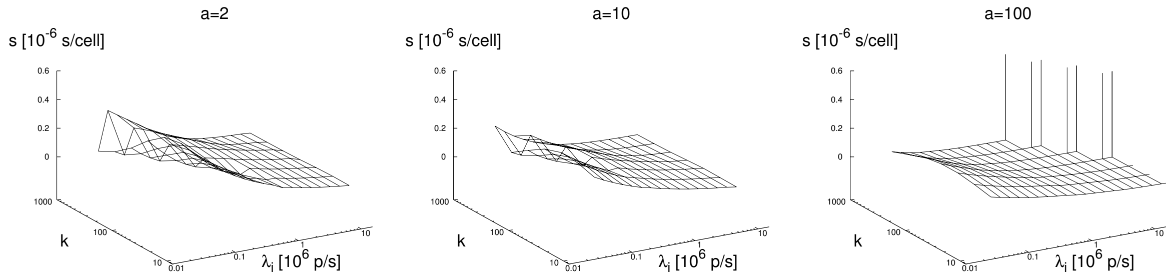


Figure 3.14: Switch port transmission speed.

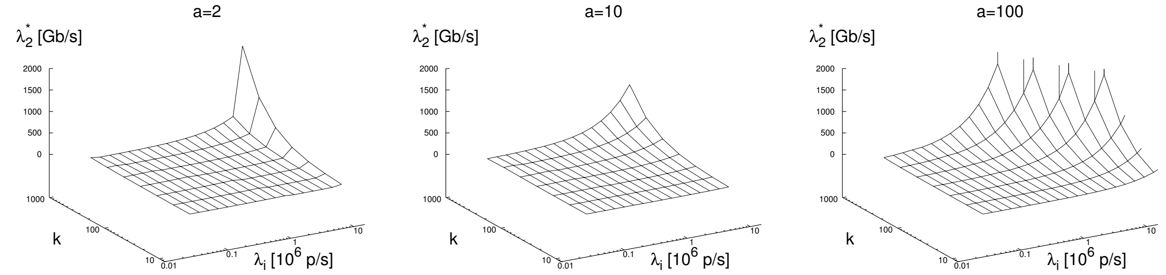


Figure 3.15: Traffic at the switch master port—the header passing overhead within the switch.

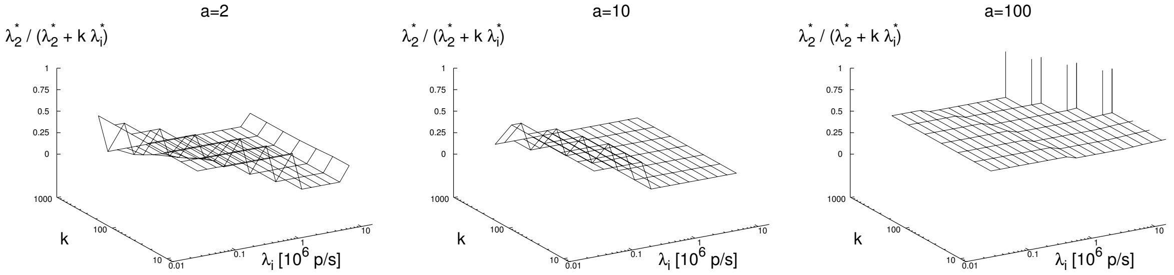


Figure 3.16: Fraction of the header passing overhead out of the total switch traffic.

is placed under increased demand to compensate the delay. For the very high-speed, many-input case, the switch is not able to cope with the demand and reaches the limit of its port transmission speed. Therefore a feasible solution for this region does not exist. Note that a similar phenomenon is only narrowly avoided in the case of $a = 2$, when the LFEs are already saturated as well.

To summarize the optimization results, we observe that for each a and b, a similar trend of dividing the problem space according to the optimum solution exists. However, the boundary line and the appropriate switch element parameters vary significantly for different a and b, which suggests that they are important factors in any router design. Note that the results are, to a large extent, also dependent on c, the ratio of the average packet size and the packet header size. If, for example, the average packet size were to decrease, the influence of the packet-header-passing overhead traffic on the system would gain a much higher significance, and vice versa.

3.5 Load sharing in a router

In this section, we outline some desired properties of router architectures. In particular, we postulate the requirements for a router that deploys a load-sharing mechanism, and reason why such mechanism is beneficial for a router system.

3.5.1 Acceptable load sharing

Our basic premise is that a router which provides for *load sharing* among the multiple Network Processors (NPUs) or forwarding engines present within the system, is able to support a greater number of interfaces, while upholding the performance guarantees. We demonstrate a load sharing–based packet processing scheme *maximizes the number of router interfaces* that can be supported with a *fixed amount of network processors of given processing power*, while keeping the advantages and avoiding the drawbacks of the router architectures discussed in Section 2.3.

The packet processing tasks are carried out by multiple distributed processors, and packets are scheduled among them according to a mapping computed at run-time. Thus, the total load of the router system is shared among the multiple processing units. The subsequent increase in processor utilization lowers the total system cost and the electricity power consumption. In addition, router fault tolerance is improved.

Our objective can be reformulated as follows: given a router, containing a set of m network processors of processing powers μ_j and given a maximum line card speed $\hat{\gamma}$, maximize the number of interfaces n that such a router can support with a performance constraint P (packet loss) $< \epsilon_p$, where ϵ_p is a given constant. We assume that any processor $j \in \{1, \ldots, m\}$ is able to process any packet.

Definition 2 provides a useful reference point for achieving the objective.

Def. 2 ***Acceptable Load Sharing.*** *Let $\lambda_j(t)$ be the packet processing load at processor j at time t and let $\lambda(t) = \sum_{j=1}^{m} \lambda_j(t)$ be the total packet processing load on the system. Let $\mu = \sum_{j=1}^{m} \mu_j$ be the total system packet processing capacity. Let us define as acceptable load sharing a scheme distributing the interfaces' load among the network processors with the following properties:*

- *if $\lambda(t) \leq \mu$, then $\forall j,\ \lambda_j(t) \leq \mu_j$, i.e., if the system* is not *overloaded, then* none *of the individual processors is overloaded,*
- *if $\lambda(t) > \mu$, then $\forall j,\ \lambda_j(t) > \mu_j$, i.e., if the system* is *overloaded, then* all *of the individual processors are overloaded.*

Generally, P (packet loss) $= \sum_{j=1}^{m}\ P\ (\lambda_j(t) > \mu_j)$. In the case of acceptable load sharing, a single processor is overloaded if and only if the entire system is overloaded, thus P' (packet loss) $= P\ (\lambda(t) > \mu) = P\ (\sum_{j=1}^{m} \lambda_j(t) > \sum_{j=1}^{m} \mu_j)$. Clearly, P' (packet loss) $\leq P$ (packet loss) and P'(packet loss) is the *minimal achievable packet loss probability.* Thus, as acceptable load sharing minimizes the packet loss probability for a given total system load, it enables to increase the total system load to the uppermost limit possible within the above packet loss probability constraint. Assuming a proportional equivalence between the number of router interfaces of given speed and the maximal total packet processing load on the system, the number of supportable interfaces is thus maximized.

3.5.2 Other system requirements

In addition to performance guarantees, a load-sharing system among parallel NPUs should possess the following properties:

Flow order preservation–packet reordering could occur if packets belonging to the same flow were processed by different NPUs. Thus, the assignment of packets to processors should either be fully deterministic with respect to flows, or should attempt to minimize the probability of packets belonging to the same flow being treated by different processors.

Absence of state information–keeping state information on assignment of concurrent flows is extremely costly in terms of memory and processing overhead. Therefore, it is highly desirable that the assignment of flows to processors can be carried out without the state information being stored.

Support for heterogeneous processors–the system must be able to support heterogeneous architectures, that is, where there are processors with various processing capacities present or where preference should be given to some processors as to the amount of requests processed.

Fault tolerance–the system must be able to adjust to a processor failure quickly and gracefully, i.e. without great disruption.

3.6 Conclusions

In this preparatory chapter, we have carried out an analysis and optimization of a router architecture in order to obtain an understanding of the influences of various system elements and packet processing paths on the system architecture. We have presented a model for analyzing a general distributed router architecture and determining its most economical variant, given a set of performance requirements.

The studies bring insight into how the individual elements influence the router architecture with a given set of constraints. We have demonstrated that for most systems, although deploying LFEs is advantageous because the switch bottleneck and the high MFE cost are avoided, putting a load-sharing method into operation is necessary in order to reach the optimal architecture. Without a load-sharing method in place, the packet processing load can not be spread over multiple processors, as is required in the case of the MFE pool.

In the final part of the chapter, we have analyzed the contribution of deploying a load-sharing method within a router system. We have defined the notion of acceptable load sharing and demonstrated the benefits in router performance if the acceptable load sharing property is upheld. Finally, we have listed other key requirements for building a multiprocessor router system.

Chapter 4

Adaptive load sharing with minimal flow disruption

4.1 Introduction

In this chapter, the key adaptive heuristic for mapping packets to processors is presented, together with proofs of the desired minimal disruption property of the adaptation.

In the initial section, we define an abstract, parameterized model of a router system, equipped with multiple Network Processors (NPUs). In the next section, we define the packet-to-NPU mapping, a computation based on which the packets are mapped to NPUs for processing. Afterwards, the problem statement for optimizing the mapping in order to approach the desired load-sharing characteristics is presented. We show the $\mathcal{NP}$-Completeness of the optimization problem and discuss the difficulties related to predicting the mix of future packet arrivals.

In the next section, we present the adaptive heuristic, based on a feedback control loop, that seeks to produce a near-optimal solution to the optimization problem. We discuss the role of various parameters in the heuristic. Finally, we prove that the adaptation possesses the minimal disruption property, the key property for achieving the desired goal of minimizing remappings.

4.2 Notation and assumptions

We consider a router model where different processors are dedicated to the data plane and to the control plane. We use the term *Network Processor (NPU)* to denote the device performing the packet processing tasks (such as address lookup, classification, filtering, etc.), that means, the processor dedicated to the data path within a router. In contrast, we denote as *Control Point (CP)* a processor that performs the router control functions such as shortest path computation, topology information dissemination or traffic engineering. Our work concentrates on issues primarily related to the data path within a router.

The router consists of n input-output line cards, m NPUs and at least one CP. With respect to NPUs we consider a heterogeneous router model, where each processor may have different processing power. Thus, by μ_j we denote the processing power of NPU j, that is, the maximum number of packet processing units an NPU j is able to carry out per time unit Δt. We denote μ the total system processing power, that is, $\mu = \sum_1^m \mu_j$.

By $\lambda_j(t)$ we denote the actual packet processing load of NPU j, that is, the amount of packet processing units carried out at NPU j during the interval $(t - \Delta t, t]$. By $\lambda(t)$ we denote the total processing load of the system within the time interval, that is, $\lambda(t) = \sum_1^m \lambda_j(t)$. By $\rho_j(t)$ we define the workload intensity of each NPU, that is, $\rho_j(t) = \lambda_j(t)/\mu_j$, and by $\rho(t)$ the total system workload intensity, $\rho(t) = \lambda(t)/\mu$.

By $\gamma_i(t)$ we denote the amount of packets that arrived at line card i in time interval $(t - \Delta t, t]$. The maximum transport capacity of each link is $\hat{\gamma}$, thus, $\forall i$, $\hat{\gamma} \geq \gamma_i(t)$.

We define the *packet information vector* $\vec{w} = (w_1, w_2, \ldots, w_{k_w})$ as the set of k_w packet fields that are examined, processed or altered within a router and that carry the information based on which the subsequent next-hop of the packet and the treatment applied to the packet within a router are determined (i.e., for example, the destination address, the source port, TTL, URL, label, etc.). We denote as W the packet information vector space, i.e. the vector space consisting of all possible values of packet information vector $\vec{w} \in W$.

A packet containing an information vector $\vec{w}$ consumes $l(\vec{w})$ processing units at an NPU. We define as *arrival vector* $\vec{a}(t) = (a_{\vec{w}_1}(t), \ldots, a_{\vec{w}_{|W|}}(t))$ a vector of size $|W|$, where the element $a_{\vec{w}}(t)$ denotes the number of packets containing the information vector $\vec{w}$ that arrived at a router during a time interval $(t - \Delta t, t\,]$. Thus $\sum_1^n \gamma_i(t) = \sum_{\vec{w}} a_{\vec{w}}(t)$ and $\lambda(t) = \sum_{\vec{w}} a_{\vec{w}}(t)\, l(\vec{w})$.

We denote as flow *identifier vector* $\vec{v} = (v_1, v_2, \ldots, v_{k_v})$ a set of predefined packet fields that do not change within a particular flow. Each v_i represents a piece of data within the packet and the integer k_v, $k_v \geq 1$, represents the number of fields contained in vector $\vec{v}$. Typically, but not necessarily, $\vec{v}$ is composed of some fields contained within the packet header. For our purposes, any predefined set of fields (or just one of them) that remain constant within a flow can serve as the identifier vector. In this work we assume that $\vec{v} \subseteq \vec{w}$. By V we denote the vector space corresponding to all the possible values of the identifier vector $\vec{v}$ (once the format of the identifier vector is established).

A typical example of an identifier vector would be the traditional flow ID, consisting of a 5-tuple of protocol number (prot), source and destination ports (SP, DP) and source and destination addresses (SA, DA), that is, in such a case, $k_v = 5$ and $\vec{v} =$ (prot, SP, SA, DP, DA). Alternatively, one could use the destination address as a unique parameter, thus $\vec{v} = (v_1) =$ (DA). In the first case, V would represent a set of all possible flow IDs, whereas in the second case, V would be equal to the protocol address space.

Let us define as *identifier persistence vector* $\vec{\Delta}(t) = (\Delta_{\vec{v}_1}(t), \ldots, \Delta_{\vec{v}_{|V|}}(t))$, $\Delta_{\vec{v}}(t) \in \{0, 1\}$ a vector that monitors the persistence of certain flow (determined by an identifier vector) within a time interval $(t - 2\Delta t, t\,]$. We consider a flow persistent if in each of the two consecutive time intervals $(t - 2\Delta t, t - \Delta t\,]$ and $(t - \Delta t, t\,]$ a packet belonging to the flow arrives. We assume that only persistent flows are vulnerable to reordering, which can occur when consecutive packets belonging to a persistent flow are processed by different processors.

We define time interval ΔT to be the maximum time a single packet spends in the system. If no packet of a flow arrives during the time interval $(t - \Delta T, t\,]$, we assume that processing a subsequent packet from the flow at any processor does not lead to reordering.

In our scenario, we assume that any processor $j \in \{1, \ldots, m\}$ is able to process any packet.

4.3 Packet-to-NPU mapping

According to the classification by Tantawy and Zitterbart [61] (see Section 2.2.2), the specific kind of parallelism employed in the load-sharing algorithm presented here would best be characterized as spatial parallelism, i.e., packets are scheduled to multiple processors, which are all capable of carrying out the same tasks (although they do not necessarily possess homogeneous processing capacity). A mapping is established between flows and processors. It is based on the CARP HRW [50] mapping, as described in Section 2.2.3, extended by an *adaptive control loop.* As in the Network Dispatcher [24] concept (see Section 2.4.3), flows are mapped to processors, yet no state information on particular flows is stored. The HRW mapping is hash-based and is thus easily computable at high speeds (as opposed to, for example, a table-based lookup or classification). The mapping possesses several advantages over other hash-based load balancing schemes–it allows to split the hashed objects into hash buckets of arbitrary size, as determined by predefined weights, and, as we prove in Section 4.5.3, a specific method for the weights' adaptation can be found, which results in only a minimal disruption of the mapping. *Optimization* and *adaptation* of the mapping is the subject of this chapter.

The mapping adaptation procedure aims to prevent individual processor overload. The design is complicated by the need to minimize the probability of packet reordering within one flow. If packets from the same flow are to be processed by different processors, packet reordering can easily occur. Therefore, packets belonging to a particular flow should be processed by the same processor.

A fully optimal mapping is not achievable for several reasons. It is not possible to monitor the full traffic characteristics in a router, including per-flow state, nor to predict the exact traffic pattern in the next time interval. Even if that were the case, one would have to solve an $\mathcal{NP}$-Complete mapping problem at run-time, as we show in Section 4.4. However, we show that the heuristic presented here, which uses

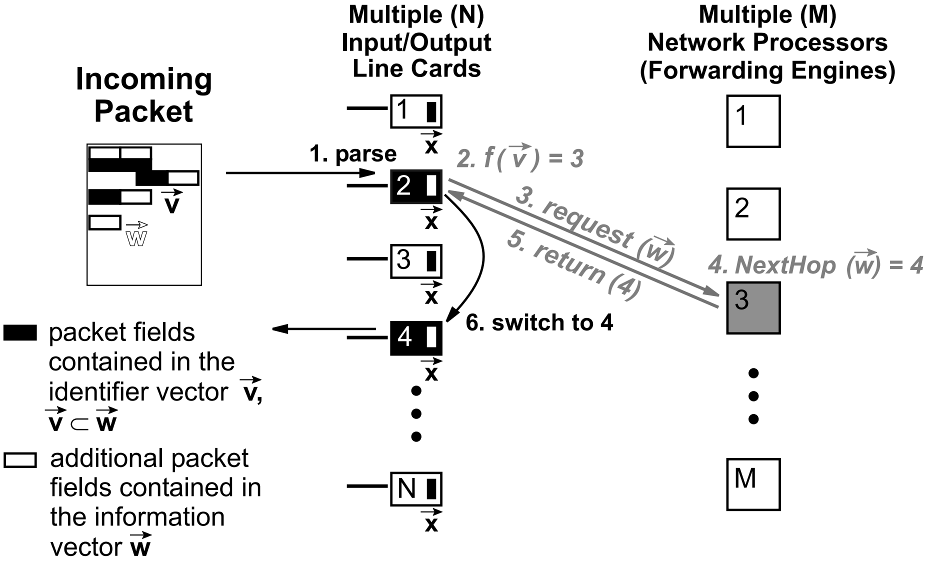

Incoming Packet
Multiple (N) Input/Output Line Cards
Multiple (M) Network Processors (Forwarding Engines)
v
w
1. parse
2. f(v) = 3
3. request (w)
4. NextHop (w) = 4
5. return (4)
6. switch to 4
x
1
2
3
4
N
M
packet fields contained in the identifier vector v, v ⊂ w
additional packet fields contained in the information vector w

Figure 4.1: **Load-sharing scheme abstraction.** Upon arrival, a packet is parsed to extract the relevant fields, the identifier vector $\vec{v}$ and the packet information vector $\vec{w}$. Then, the packet is buffered, the mapping $f(\vec{v})$ is computed and the packet information vector $\vec{w}$ is sent for resolution to NPU j, $f(\vec{v}) = j$. At NPU j, the vector $\vec{w}$ is processed and the resolution information is returned to the requesting unit. The packet is then switched to the correct output port and the corresponding packet alterations or manipulations, based on the resolution results, are applied.

aggregate traffic monitoring as feedback, stays within small bounds from the optimal solution.

The basis of our load-sharing scheme is that the load of each input (ingress traffic arriving at a line card) is distributed for processing among the NPUs using a *deterministic* mapping f (see Figure 4.1). The mapping f is computed over the *identifier vector* $\vec{v}$. The computation $f(\vec{v}) = j$ determines the particular NPU j to which the packet is mapped for processing. The function $f(\vec{v})$, $f : V \to \{1, 2, \ldots, m\}$, splits the vector space V into m exclusive subspaces V_j. Packets from a particular subspace are all mapped to the same processor.

Upon arrival of a packet at an input, the packet is parsed to extract the fields relevant for packet processing, i.e., the identifier vector $\vec{v}$ and the packet information vector $\vec{w}$. The packet is buffered, the mapping $f(\vec{v})$ is computed and the packet information vector $\vec{w}$ is then sent for resolution to NPU j, $f(\vec{v}) = j$.

At NPU j, the packet information vector $\vec{w}$ is processed and the resolution information about the treatment to be applied to the packet (next hop, outgoing switch port, QoS applied) is returned to the requesting unit. Then, the packet is switched to the correct outgoing port and the corresponding packet alterations or manipulations, based on the resolution results, are applied (this may mean, for example, applying certain QoS, attaching an MPLS label or splicing with another TCP connection).

The mapping f we propose for such a purpose is based on the robust hash routing scheme (alternatively called highest random weight (HRW) mapping) presented in [62] and extended in [50].

Def. 3 ***Packet-to-NPU (HRW) Mapping f:*** *Let $g(\vec{v}, j)$ be a pseudo-random function $g : V \times \{1, 2, \ldots, m\} \to (0, 1)$, i.e., we assume $g(\vec{v}, j)$ to be a random variable in $(0, 1)$ with uniform distribution. Let a packet arrive at an input i, carrying an identifier vector $\vec{v} \in V$. The mapping $f(\vec{v})$ is then computed as follows:*

$$f(\vec{v}) = j \tag{4.1}$$

$$\Leftrightarrow$$

$$x_j \, g(\vec{v}, j) = \max_{k \in \{1,\ldots,m\}} x_k \, g(\vec{v}, k), \tag{4.2}$$

where $x_j \in \mathbb{R}^+$ is a weight multiplier assigned to each NPU.

The weights $\vec{x} = (x_1, \ldots, x_m)$, as described in Section 2.2.3, are in a 1-to-1 correspondence with the partitioning vector $\vec{p} = (p_1, \ldots, p_m)$, which determines the fraction of request object space (the identifier vector space V, in our case) assigned for processing to each NPU, i.e., $p_j = |V_j|/|V|$.

The following HRW mapping properties (see Section 2.2.3) are particularly useful for the purpose of flow-to-processor mapping:

Load balancing over the request object space. The method provides load balancing even for the heterogeneous case. That is extremely useful for the ability to support processors of heterogeneous processing capacities because the mapping weights allow the fraction of load mapped to a particular processor to be controlled.

Minimal disruption. In the case of a processor failure, removal or addition, the number of request objects that are re-mapped to another destination is minimal. This property is useful for providing *fault tolerance* (if a particular processor fails, only flows mapped to that processor are affected).

It is vital to realize, though, that even if the algorithm provides load balancing over the request object space, it is *not adaptive* and therefore potentially vulnerable to locality in the input traffic.

Note that the advantages of minimal disruption property are not limited to the above special cases. In Section 4.5.3 we show that by a similar line of proof as in [62], the minimal disruption property holds as well for *certain special kinds of adjustments* of the mapping weights. We exploit that fact when carrying out the mapping adaptation in order to *minimize the amount of flow remappings* caused by the adaptation.

For the load-sharing purposes in general, there is no need for the mapping f to be identical at all line cards. In fact, a different mapping can be used at each line card, for example, at line card i, a mapping $f_i(\vec{v})$ could be computed using function g_i of the form $g_i(\vec{v}, j) = g(\vec{v}, i + j)$. However, our scheme does require that the weights vector $\vec{x}$ be identical at each card.

4.4 Optimization problem statement

Load sharing among multiple processors can become very inefficient if attention is not paid to keeping the individual processor load under control. The goal of the adaptation is to prevent undesirable effects, mainly, processor overload and a consequent packet loss. It may not be obvious how such effects can occur when, as claimed, the HRW mapping provides load balancing. However, it is important to note that it provides load balancing over the *request object space*, i.e., in our case, the identifier vector space V. In contrast, the loads due to the actual traffic received at the router input ports may by no means be distributed uniformly over this request object space, but rather will exhibit certain locality patterns. That means that in spite of the load-balancing property, mapping f can potentially lead to grossly imbalanced load distributions. For such cases, the mapping must be adjusted to account for the non-uniform load distribution in the received traffic. As thus the mapping f now changes with time, we define $f_{(t)}(\vec{v}) : V \rightarrow \{1, 2, \ldots, m\}$ as the instance of f at time t.

The objective of the control loop is to prevent over-utilization of a single processor when the system is under-utilized or, vice-versa, to prevent under-utilization of a single processor when the system is over-utilized. At the same time, we aim to minimize the amount of packet-to-NPU remappings. Assuming that the mapping f is being adjusted periodically in time intervals Δt, the objective for the adjustment at time $t - \Delta t$ can be formulated as the following optimization problem:

Def. 4 ***NPU load-sharing optimization problem:***

$$\max \sum_{\vec{v} \in V} \Delta_{\vec{v}}(t) \sum_{j \in M} 1\{f_{(t-\Delta t)}(\vec{v}) = j\} * 1\{f_{(t)}(\vec{v}) = j\}, \tag{4.3}$$

with constraints:

$$\textit{if } \rho(t) \leq 1 \quad \Rightarrow \quad \lambda_j(t) \leq \mu_j, \forall j, \tag{4.4}$$

$$\textit{if } \rho(t) > 1 \quad \Rightarrow \quad \lambda_j(t) \geq \mu_j, \forall j, \tag{4.5}$$

where

$$\lambda_j(t) = \sum_{\vec{v} \in V} 1\{f_{(t)}(\vec{v}) = j\} \sum_{\vec{w} \supseteq \vec{v}} a_{\vec{w}}(t)\, l(\vec{w}). \tag{4.6}$$

Note that this problem statement is only useful for defining the objective of our method, but not for computing the actual solution $f_{(t)}$. In order to be able to carry out the optimization described in Def. 4, one would have to know already at time $t - \Delta t$ both $\Delta_{\vec{v}}(t)$, which would require to maintain a huge amount of state information, as well as $a_{\vec{w}}(t)$, which can only be speculatively predicted. Furthermore, even if such knowledge were available, one would still have to solve an $\mathcal{NP}$-complete problem, as Def. 4 is an Integer Linear Programming optimization. Thus, heuristics, such as the one presented below, are typically used, yet the above definition remains useful for setting the objective and evaluating the quality of the solution ex-post.

Note that in order to connect the formula for minimizing the packet-to-NPU remappings to minimizing the number of packets reordered, the adaptation interval Δt must satisfy $\Delta t \geq \Delta T$. Then, during the interval a flow is vulnerable to reordering, ΔT, a packet flow can be re-mapped at most once and thus the minimization of the number of remappings applies to minimizing the number of re-orderings as well.

4.5 Adaptive load-sharing heuristic

4.5.1 Adaptation algorithm

The adaptation scheme works in the following general way (see Figure 4.2): periodically, the CP gathers information about the workload intensity of the NPUs. If an adaptation threshold is exceeded, the CP adjusts the weights of the mapping f. The new multiplicative weights vector $\vec{x}$ is then downloaded to the NPUs. Let $\vec{x}(t)$ be the instance of weights' vector $\vec{x}$ used to compute $f(t)$ at time t.

In order to evaluate the status of individual processors, we need a processor workload intensity indicator. For that purpose, we introduce a smoothed, low-pass *filtered processor workload intensity* measure $\bar{\rho}_j(t)$ of the form

$$\bar{\rho}_j(t) = \frac{1}{r}\,\rho_j(t) + \frac{r-1}{r}\,\bar{\rho}_j(t-\Delta t), \tag{4.7}$$

where r is an integer constant. A similar filtered measure for total system workload intensity is introduced as $\bar{\rho}(t) = \frac{1}{r}\,\rho(t) + \frac{r-1}{r}\,\bar{\rho}(t-\Delta t)$. The filtering is done to reduce the influence of short-term load fluctuations and to obtain information about the *trend* in processor workload intensity.

The adaptation algorithm consists of two parts (see Figure 4.3): the *triggering*

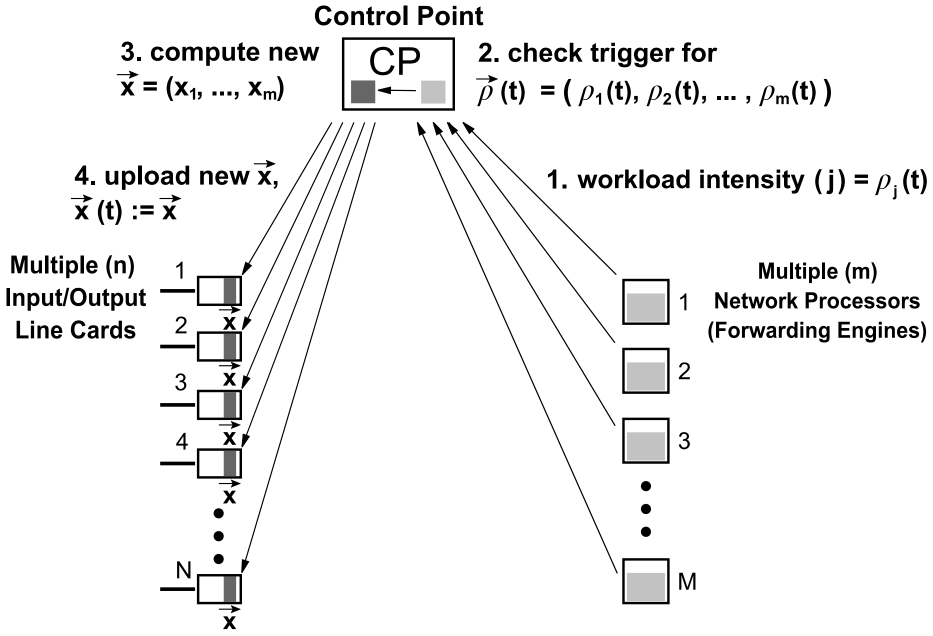


Figure 4.2: **Load sharing with feedback.** Periodically, the CP gathers information about the workload intensity of the NPUs $\vec{\rho}(t) = (\rho_1(t), \rho_2(t), ..., \rho_m(t))$. If the adaptation is triggered, the CP adjusts the multiplicative weights vector $\vec{x}$ and the new vector is then downloaded to the NPUs.

policy, which specifies the conditions to act, and the *adaptation policy*, which specifies how to act. A trigger is periodically evaluated and, based on the result, specific action is taken.

4.5.2 Triggering policy

We introduce a dynamic *workload intensity threshold* $\epsilon'_\rho(t)$ defined as

$$\begin{aligned} \epsilon'_\rho(t) &= \bar{\rho}\ (t) + \frac{1}{2}\ (1 - \bar{\rho}\ (t)) && (4.8) \\ &= \frac{1}{2}\ (1 + \bar{\rho}\ (t)). && (4.9) \end{aligned}$$

Thus the dynamic workload intensity threshold is positioned midway between the current filtered total system workload intensity $\bar{\rho}(t)$ and workload intensity of 1. The

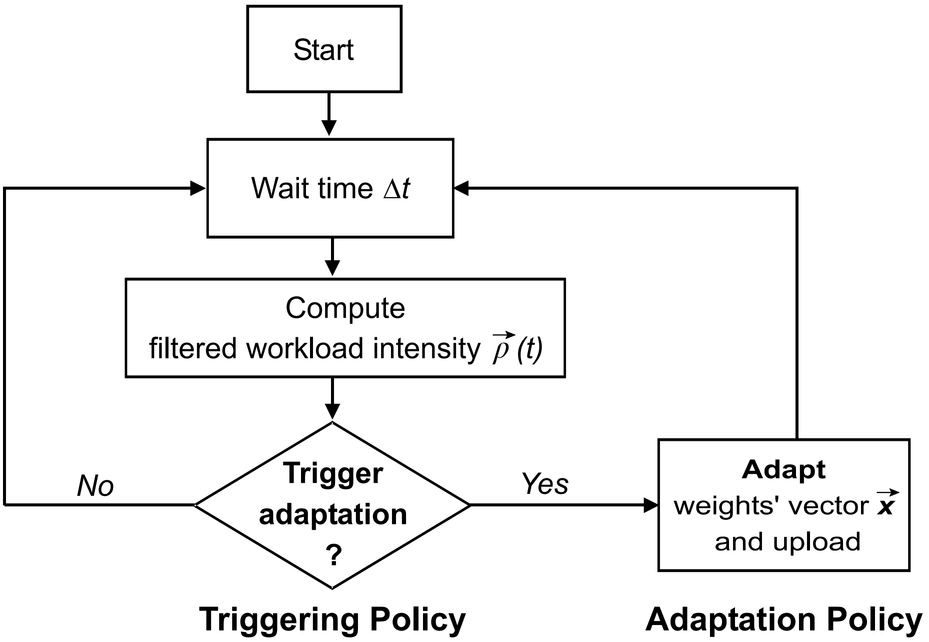


Figure 4.3: Adaptation algorithm.

closer the total system workload intensity approaches 1, the higher the likelihood of violating the acceptable load-sharing bounds and therefore the tighter the threshold follows the total system workload intensity.

During time intervals when the total system workload intensity $\bar{\rho}(t)$ remains in the vicinity of 1, the value of the workload intensity threshold may be too close to $\bar{\rho}(t)$ to provide a meaningful threshold for adaptation. To prevent such cases, we introduce a form of *hysteresis* into the threshold computation by defining a fixed threshold in the close vicinity of 1.

Let $\epsilon_h > 0$ be a fixed hysteresis bound, which prevents adaptation from being carried out within the interval $((1-\epsilon_h)\,\bar{\rho}(t),\ (1+\epsilon_h)\,\bar{\rho}(t))$. Typically, ϵ_h is set to a value close to 0, for example 0.01, thus preventing adaptation when the load stays within 1 percent of the total system workload intensity.

A dynamic triggering threshold $\epsilon_\rho(t)$, which combines the workload intensity threshold $\epsilon'_\rho(t)$ with the hysteresis bound, is thus set for determining the amount of over- (or under-) utilization allowed at one processor:

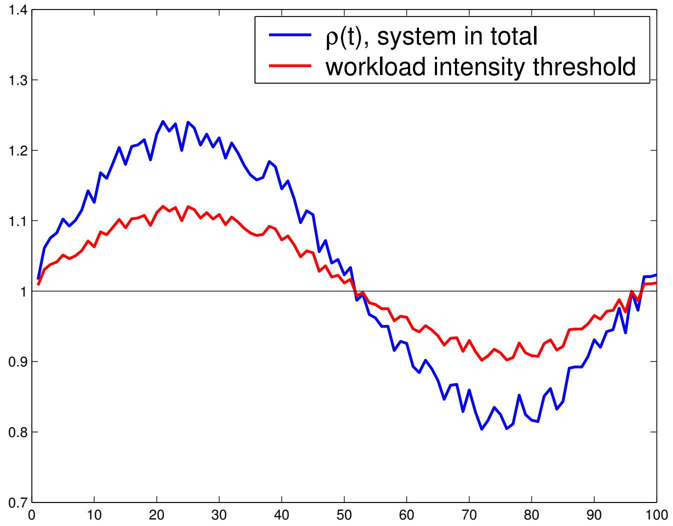


Figure 4.4: Example of the dynamic workload intensity threshold as a function of a hypothetical filtered total system workload intensity.

Def. 5 ***Triggering Threshold $\epsilon_\rho(t)$:*** *Let the dynamic workload intensity threshold $\epsilon'_\rho(t)$ and the hysteresis bound ϵ_h be defined as above. Then the triggering threshold is defined (according to whether the system in total is over- or under-utilized) as follows:*

$$\epsilon_\rho(t) = \max\ (\epsilon'_\rho(t),\ (1+\epsilon_h)\,\bar{\rho}(t)), \quad \bar{\rho}(t) \leq 1, \tag{4.10}$$

$$\epsilon_\rho(t) = \min\ (\epsilon'_\rho(t),\ (1-\epsilon_h)\,\bar{\rho}(t)), \quad \bar{\rho}(t) > 1. \tag{4.11}$$

The result of the comparison of the filtered workload intensity to the threshold then acts as a trigger for the adaptation to start. An appropriate trigger is again chosen according to whether the system in total is over- or under-utilized:

$$\bar{\rho}(t) \leq 1 \quad \Rightarrow \quad \text{if } (\epsilon_\rho(t) < \max_j \bar{\rho}_j(t)) \text{ then } \textit{adapt}$$

$$\bar{\rho}(t) > 1 \quad \Rightarrow \quad \text{if } (\epsilon_\rho(t) > \min_j \bar{\rho}_j(t)) \text{ then } \textit{adapt}.$$

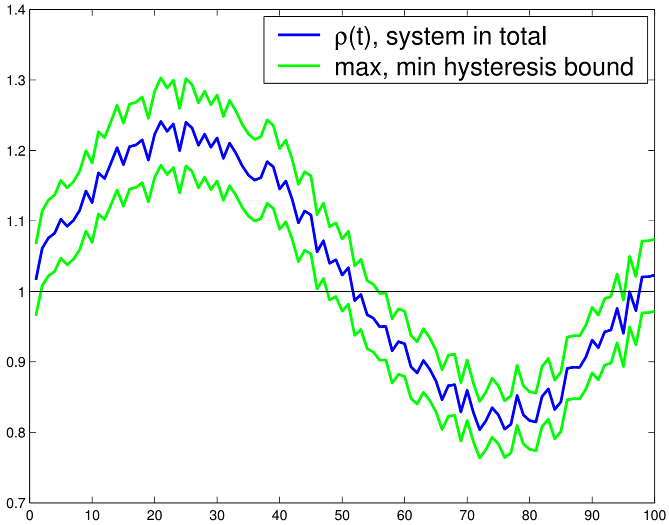


Figure 4.5: Example of the hysteresis bound as a function of a hypothetical filtered total system workload intensity.

4.5.3 Minimal disruption

Lemma 1 and Theorem 2 provide the theoretical basis for the mapping's adaptation:

Lemma 1 *Let* $\alpha \in \mathbb{R}^+$, $\alpha \neq 1$. *Let* A, B *be two nonempty, mutually exclusive subsets of* $M = \{1, \ldots, m\}$, $M = A \cup B$. *Let* f, f' *be two HRW mappings using identical pseudo-random function* $g(\vec{v}, j)$, *but differing in the weight vectors* $\vec{x} = (x_1, \ldots, x_m)$ *and* $\vec{x}\,' = (x'_1, \ldots, x'_m)$ *as follows:*

$$x'_j = \alpha\, x_j, \quad j \in A, \tag{4.12}$$

$$x'_j = x_j, \quad j \in B. \tag{4.13}$$

Let p_j *and* p'_j, *denote the fraction of request object space mapped to node* j *using the HRW mapping with weights* $\vec{x}$ *and* $\vec{x}\,'$, *respectively. Then, if* $\alpha < 1$,

$$p'_j \leq p_j, \quad j \in A \tag{4.14}$$

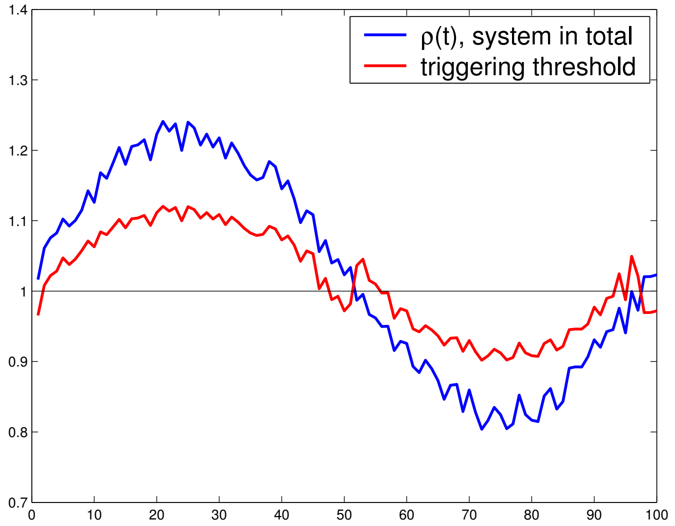


Figure 4.6: Example of the triggering threshold as a function of a hypothetical filtered total system workload intensity.

$$p'_j \quad \geq \quad p_j, \quad j \in B. \tag{4.15}$$

and, conversely, if $\alpha > 1$,

$$p'_j \quad \geq \quad p_j, \quad j \in A \tag{4.16}$$

$$p'_j \quad \leq \quad p_j, \quad j \in B. \tag{4.17}$$

Proof: We first prove the inequality (4.14) by contradiction. Assume that $\exists j_0 \in A$ such that $p'_{j_0} > p_{j_0}$. It means that there exists at least one identifier vector $\vec{v}_0$, for which $f'(\vec{v}_0) = j_0$, yet $f(\vec{v}_0) \neq j_0$.

As $f'(\vec{v}_0) = j_0$, we have $x'_j\, g(\vec{v}_0, j_0) = \max_{k \in M}\ x'_k\, g(\vec{v}_0, k)$. But then:

$$\begin{aligned} x_{j_0} g(\vec{v}_0, j_0) &= \frac{1}{\alpha}\ x'_{j_0} g(\vec{v}_0, j_0) \\ &\geq \frac{1}{\alpha}\ x'_k g(\vec{v}_0, k) \\ &= x_k g(\vec{v}_0, k), \qquad \forall k \in A; \end{aligned}$$

$$\begin{aligned} x_{j_0} g(\vec{v}_0, j_0) &= \frac{1}{\alpha}\, x'_{j_0} g(\vec{v}_0, j_0) \\ &\geq \frac{1}{\alpha}\, x'_k g(\vec{v}_0, k) \\ &= \frac{1}{\alpha}\, x_k g(\vec{v}_0, k) \\ &\geq x_k g(\vec{v}_0, k), \qquad \forall k \in B. \end{aligned}$$

Therefore, $x_{j_0} g(\vec{v}_0, j_0) = \max_{k \in M} x_k g(\vec{v}_0, k)$ and thus $f(\vec{v}_0) = j_0$, which is a contradiction to our assumption.

Inequality (4.15) can be proved in a symmetrical way, as well as the case of $\alpha > 1$. □

Equality in inequalities (4.14)-(4.17) is an extreme case, which can only take place if α is so close to 1 that the weights $\vec{x}$ change so little that the change does not affect any single identifier vector.

Note that given the complex relationship among vectors $\vec{x}$ and $\vec{p}$ (see [50]), it is hard to say more in general about the effects of direct adjustments of $\vec{x}$.

Theorem 2 (Minimal disruption) *Let* $\alpha \in \mathbb{R}^+$*. Let* A, B *be two nonempty, mutually exclusive subsets of* $M = \{1, \ldots, m\}$*,* $M = A \cup B$*. Let* f*,* f' *be two HRW mappings using identical pseudo-random function* $g(\vec{v}, j)$*, but differing in the weight vectors* $\vec{x} = (x_1, \ldots, x_m)$ *and* $\vec{x}\,' = (x'_1, \ldots, x'_m)$ *as follows:*

$$x'_j = \alpha\, x_j, \quad j \in A, \tag{4.18}$$

$$x'_j = x_j, \quad j \in B. \tag{4.19}$$

Let p_j *and* p'_j *denote the fraction of request object space mapped to node* j *using the HRW mapping with weights* $\vec{x}$ *and* $\vec{x}\,'$*, respectively. Then, the fraction of request object space mapped to two different nodes by the two mappings is equal to* $\frac{1}{2}\sum_j |p_j - p'_j|$*, or, in other words, the amount of request objects mapped by the two mappings to different destinations is minimal.*

Proof: The case of $\alpha = 1$ is trivial. Let $\alpha < 1$. We prove that for each node j, exactly $|p_j - p'_j|\,|V|$ objects have changed the mapping. The proof is divided into two parts:

1. $j \in A$: from Lemma 1 we know that $p'_j \le p_j$. Let us show that all objects mapped to j by f' are also mapped to j by f by contradiction: assume that there exists at least one identifier vector $\vec{v}_0$, for which $f'(\vec{v}_0) = j$, yet $f(\vec{v}_0) \neq j$. But, if $f'(\vec{v}_0) = j$, it means that $x'_j\, g(\vec{v}_0, j) = \max_k \; x'_k\, g(\vec{v}_0, k)$ and therefore

$$\begin{aligned} x_j g(\vec{v}_0, j) &= \frac{1}{\alpha}\, x'_j g(\vec{v}_0, j) && (4.20)\\ &\ge \frac{1}{\alpha}\, x'_k g(\vec{v}_0, k) && (4.21)\\ &\ge x_k g(\vec{v}_0, k), \qquad \forall k \in M. && (4.22) \end{aligned}$$

   Thus, $x_j\, g(\vec{v}_0, j) = \max_k \; x_k\, g(\vec{v}_0, k)$ and $f(\vec{v}_0) = j$, which is a contradiction to our assumption. As all objects mapped to j by f' are also mapped to j by f, the amount of request objects in which the two mappings differ at node j is equal to the fraction $|p_j - p'_j|$ of request object space.

2. $j \in B$: from Lemma 1 we know that $p'_j \ge p_j$. Let us show that all objects mapped to j by f are also mapped to j by f' by contradiction: assume that there exists at least one identifier vector $\vec{v}_0$, for which $f(\vec{v}_0) = j$, yet $f'(\vec{v}_0) \neq j$. But, if $f(\vec{v}_0) = j$, it means that $x_j\, g(\vec{v}_0, j) = \max_k \; x_k\, g(\vec{v}_0, k)$ and therefore

$$\begin{aligned} x'_j g(\vec{v}_0, j) &= \alpha\, x_j g(\vec{v}_0, j) && (4.23)\\ &\ge \alpha\, x_k g(\vec{v}_0, k) && (4.24)\\ &\ge x'_k g(\vec{v}_0, k), \qquad \forall k \in M. && (4.25) \end{aligned}$$

   Thus $x'_j\, g(\vec{v}_0, j) = \max_k \; x'_k\, g(\vec{v}_0, k)$ and $f'(\vec{v}_0) = j$, which is a contradiction to our assumption. As all objects mapped to j by f are also mapped to j by f', the amount of request objects in which the two mappings differ at node j is equal to the fraction $|p_j - p'_j|$ of request object space.

Thus, the two mappings differ by $|p_j - p'_j|\,|V|$ vectors at each node, which leads to a fraction of $\frac{1}{2}\sum_j |p_j - p'_j|$ difference in total.

The proof for $\alpha > 1$ is symmetrical. □

It is important to note that the minimal disruption property would not generally hold for the adaptation if the weights' adjustment were not carried out by the *same constant multiplier*, as then the inequalities (4.21) and (4.24) would not necessarily hold for all $k \in M$.

The minimal disruption property of the adaptation using a single multiplier is illustrated on an example with three NPUs in Fig. 4.7.

4.5.4 Adaptation policy

We propose a simple scheme for the periodic adaptation, operating directly on the weights' vector $\vec{x}$. As the minimal disruption property is crucial for minimizing the amount of remappings, Lemma 1 and Theorem 2 serve as a background for designing the adaptation of vector $\vec{x}$ to be carried out by a single, constant multiplier:

Def. 6 ***Weights-Vector $\vec{x}$ Adaptation:*** *Let $\bar{\rho}(t) \leq 1$. Assuming that the trigger condition ($\epsilon_\rho(t) < \max_j \bar{\rho}_j(t)$) is satisfied, let*

$$c(t) = \left(\frac{\epsilon_\rho(t)}{\min\left\{\bar{\rho}_j(t) \mid \bar{\rho}_j(t) > \epsilon_\rho(t)\right\}} \right)^{1/m}. \tag{4.26}$$

Then

$$x_j(t) := \quad c(t)\, x_j(t - \Delta t), \quad \bar{\rho}_j(t) > \epsilon_\rho(t), \tag{4.27}$$

$$x_j(t) := \quad x_j(t - \Delta t), \quad \bar{\rho}_j(t) \leq \epsilon_\rho(t). \tag{4.28}$$

Conversely, the adaptation for the case of $\bar{\rho}(t) > 1$ is performed in a symmetrical manner.

Thus, in the case that the system is under-utilized, the presented adaptation *lowers the weights for the exceedingly* (with respect to a threshold) *over-utilized processors*, whereas weights for others remain unchanged. Conversely, if the system in total is over-utilized, the adaptation *raises the weights for the exceedingly* (with respect to a threshold) *under-utilized processors*. The lowering or raising of weights is carried

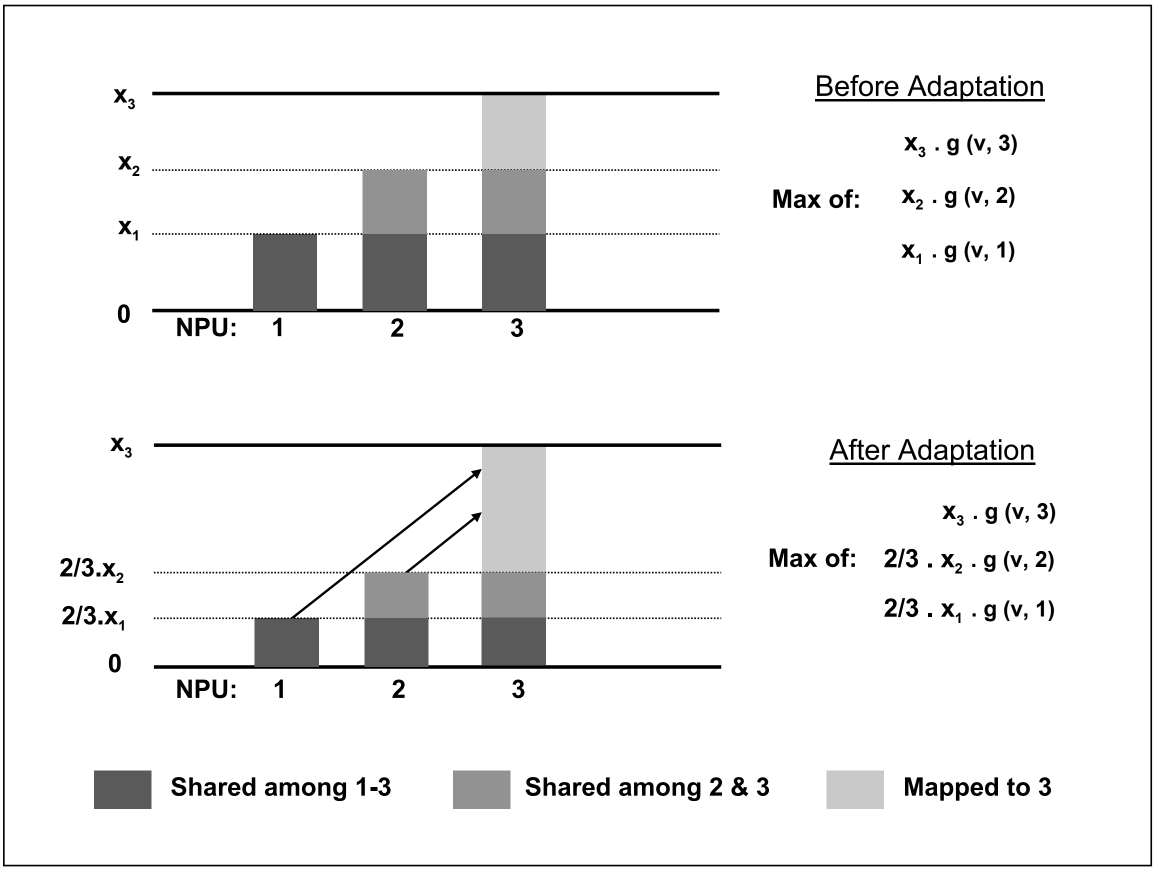


Figure 4.7: **Minimal disruption property of the adaptation.** There are three NPUs (No. 1-3) present in the system, and weights of two of the NPUs (No. 1-2) are lowered by a factor of 2/3. Objects migrate from the adjusted NPUs (No. 1-2) to the unadjusted NPU (No. 3) only. The figure depicts the hash score range per-processor, the sharing of the hash-score space among the processors and the effects of the adjustment on the hash-scores.

out proportionally, either to the minimal workload intensity $\bar{\rho}_j(t)$ which exceeds the threshold $\epsilon_\rho(t)$, or to the maximal workload intensity $\bar{\rho}_j(t)$ which remains below the threshold $\epsilon_\rho(t)$.

The factor $1/m$ in the exponent of $c(t)$ represents the effects of the number of processors present–less aggressive adjustment is needed in the case of more processors. Alternatively, the exponent of the form of $1/\log_2(m)$ can be used, making the root computation less complex. The effects of using either exponent are evaluated in

Sections 5.5.3 and 5.5.4.

4.6 Conclusions

In this chapter, we have introduced a technique how to provide load sharing in a multiprocessor system that carries out the packet processing tasks within a network node. The method is a heuristic that seeks to stay within close vicinity from the optimal solution. The principal part is a fast computable, hash-based mapping, which assigns packet flows to processors. The technique does not require to maintain state information on the flow-to-processor mapping.

The mapping is coupled with an adaptation discipline that continuously evaluates the distance from the optimal solution and, if a threshold is exceeded, adjusts the mapping parameters proportionally to the distance (error) from the optimal solution. We prove, by theoretical means, that the adaptation possesses the key *minimal disruption* property, which leads to minimizing the probability of packets belonging to one flow being processed by different processors.

Chapter 5

Method validation

5.1 Introduction

In this chapter, we validate the adaptive load-sharing method, as presented in the previous chapter, using an extensive set of simulations.

First, we present results using a simple Java-based simulator of the multiprocessor system and of the incoming traffic to verify the basic concept of the method. A simple traffic generation technique has been used to test the load spreading and the correct weights' adjustment by the adaptive control loop.

Next, we present results using a more sophisticated, Matlab-based implementation of the multiprocessor system model that uses generated traffic as input. The traffic generator, Matlab-based as well, aims to replicate the key characteristics of the real-life Internet traffic, like the flow length distribution and the source address distribution. We test the influence of various parameters, or factors, on the performance of the method.

Finally, we present some general conclusions about the adaptive load-sharing method performance.

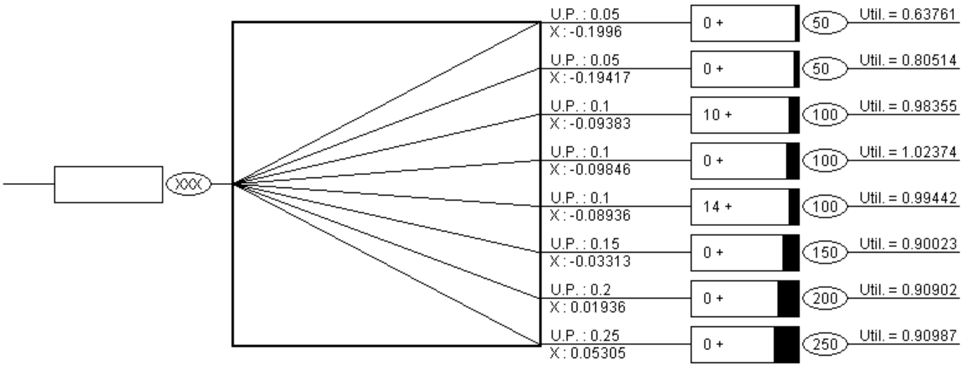


Figure 5.1: The simple load-sharing system simulator.

5.2 Simple simulator—proof of concept

We present some initial results of executing the mapping and adaptation algorithms using a simple, Java-based simulator. The system setup is depicted in Fig. 5.1.

The following are the assumptions used in the simulations:

- We assume a system with eight outgoing links that act as the processing units. There is an output queue (*OutQueue*) in front of each link, storing packets that could not yet be forwarded. All the queues have different service rates (the link speeds) and are considered infinite. The load percentages, p_j, are: [5%, 5%, 10%, 10%, 10%, 15%, 20%, 25%];

- Incoming packets contain 8 random numbers uniformly distributed within [0, 1]. All the incoming packets are stored in and extracted for forwarding from a large FIFO queue (*InQueue*);

- The $UserP_i$ is a fixed value describing the expected percentage of packets a user wants to be forwarded on link i. *PktProc* is the maximum number of packets that can be forwarded altogether over all the links in 1 second. Thus, $Rate_i = PktProc \cdot UserP_i$ is the expected number of forwarded packets per second on link i;

- The traffic generator can be shaped by the user by setting three input parameters: *average*, *max* and *PBurst*. *average* is defined as a percentage of *PktProc*. The number of queued packets at the ingress queue per simulation second is the following:

$$pkts = \begin{cases} average \cdot PktProc & \text{with prob } 1 - PBurst \\ average \cdot PktProc + U[-max, max] & \text{with prob } PBurst \end{cases}$$

- Time is discrete, and each second the simulator executes the triggering policy by evaluating where the incoming packets were forwarded. The weights' adaptation, if triggered, is then performed instantaneously.
- The filtering constant r is set to $r = 3$ and the hysteresis bound ϵ_h is set to $\epsilon_h = 0.01$.

Simulation results

At every iteration, the queue occupancy, the number of packets forwarded to a specific link, the utilization and the weights of that link are monitored. The following list summarizes the response variables:

- Packets received standard deviation:

$$Std(t) = \frac{\sum_{i=1}^{N=8} (InQueue[i].forw(t) - InQueue[i].Rate(t))^2}{N} \tag{5.1}$$

- Per second occupancy of all queues (in packets)
- Per second sum of the packets in all queues
- Per second workload intensity of all links
- Per second workload intensity of the entire system

- Per second weights value.

Here we present the results of two simulations. We have tested the system model near the critical threshold $\left(\rho(t) = \frac{\sum \lambda_i(t)}{\sum \mu_i(t)} \approx 0.95\right)$ to observe the system behavior when close to congested. Then we observed the behavior of the system when staring with incorrect initial weights.

Note that all data are merely indicative because the simulator is only a rough approximation of the real system. For each simulation we will verify whether

- the queues' occupancy does not grow explosively;
- the system and link workload intensity are close to 1;
- the standard deviation attains some acceptable values;
- the weights' values do not oscillate.

5.2.1 Potentially congested case

This 300-sec simulation subjects the system model to load that could result in congesting the load-sharing device. We set *PktProc* to 1000, and each second we generate

$$pkts = \begin{cases} 950 & \text{with prob } 80\% \\ 950 + U[-50, 50] & \text{with prob } 20\% \end{cases}$$

This scenario clearly harbors a possible congestion. We want to verify what happens when the difference between the number of packets forwarded to link j and the rate of that link is about constant and nearly 0. If the load balancer forwards the packets badly (e.g. too many packets to slower or congested links) we should observe some queue filling effect. However, this phenomenon does not appear, as on average only 50 packets are queued, with peaks of 100 packets (Fig. 5.4). Moreover, system workload intensity (Fig. 5.5) does not oscillate and always remains in the range [-0.97, +1.03]. Likewise, the standard deviation (Fig. 5.6) is always in the range of acceptable values. Finally the weights (Fig. 5.3) attain values in the range $[initial\_value - 0.02, initial\_value + 0.02]$ and we observe no apparent oscillation.

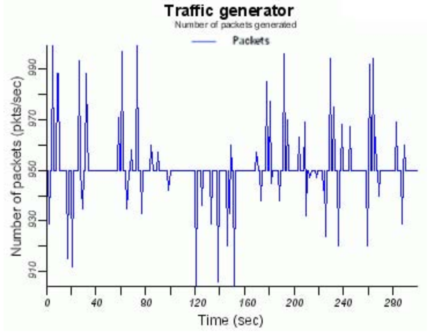


Figure 5.2: Number of packets at the ingress queue

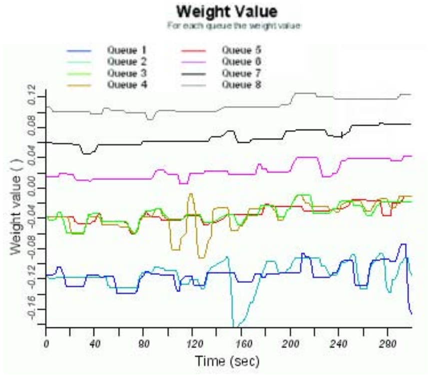


Figure 5.3: Evolution of the weights' value

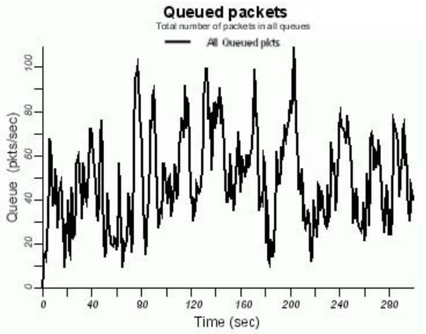


Figure 5.4: Total number of packets queued in the output queues

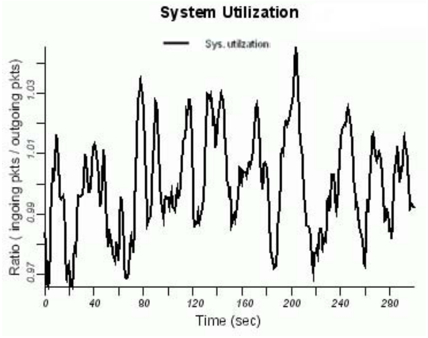


Figure 5.5: System workload intensity

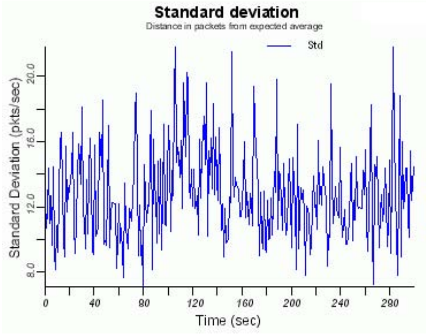


Figure 5.6: Standard deviation

5.2.2 Wrong initial weights case

This 300-sec simulation shows the behavior of the system model when the initial weights are set incorrectly ($\forall i,\ x_i = 0$). The system *converges automatically towards the correct set of weights.* We set *PktProc* to 1000, and each second we generate

$$pkts = \begin{cases} 800 & \text{with prob } 80\% \\ 800 + U[-50, 50] & \text{with prob } 20\% \end{cases}$$

We observe that the wrong initial weights ($x_i = 0$) initially produce inappropriate traffic splitting. There are two links in particular (the slowest ones) that can not cope with the assigned load, and, consequently, a lot of packets are queued (Fig 5.8). But the algorithm reacts quickly by adjusting the weights. After the first 30 seconds of simulation (this means that the adaptation is performed 30 times), the weights' values tend asymptotically to the correct ones.

Moreover, we observe that after $t = 30$, the initially large discrepancies in per-link workload intensities Fig. 5.10 settle towards a realistic value slightly below 1. Finally, note that the strong decrease in the standard deviation means that the control loop has a negative feedback, which leads to reducing the queues occupancy.

5.3 Realistic traffic generation

We have used the MATLAB v.6 environment on an IBM PC Pentium III with Microsoft Windows 2000 machine to simulate a model of a router with multiple NPUs and line cards.

For router input, we have used generated traffic. The parameters for generating the per-interface traffic were approximated from OC-3 traces statistics gathered in [2], [45] and [63] and approximated to OC-192 speed by shortening the time intervals proportionally, i.e., 1 second of the monitored OC-3 traffic corresponds to 15 ms in our OC-192-like traces. Note that this transformation is a simplification from reality, since the scaled traces would differ not only along the time dimension, but the per-flow data volume, the multiplexing effects and the packet inter-arrival times would

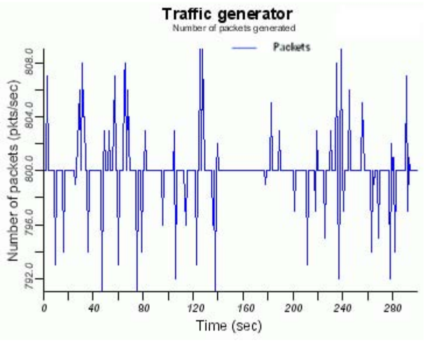


Figure 5.7: Number of packets at the input queue.

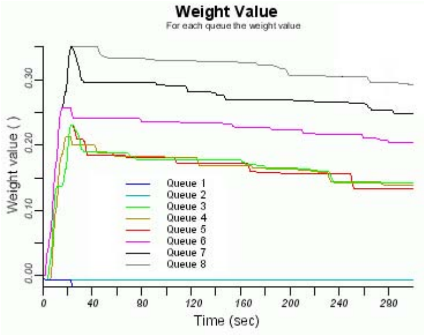


Figure 5.8: The evolution of the weights' value.

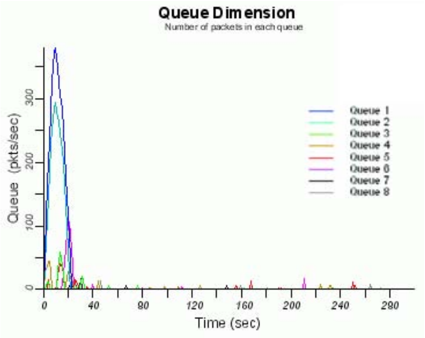


Figure 5.9: Queue occupancy.

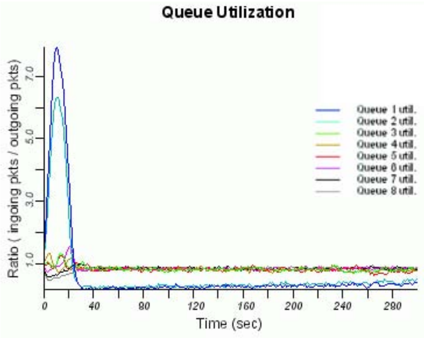


Figure 5.10: Link workload intensity.

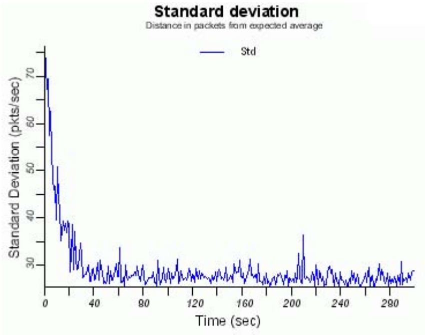


Figure 5.11: Standard deviation function.

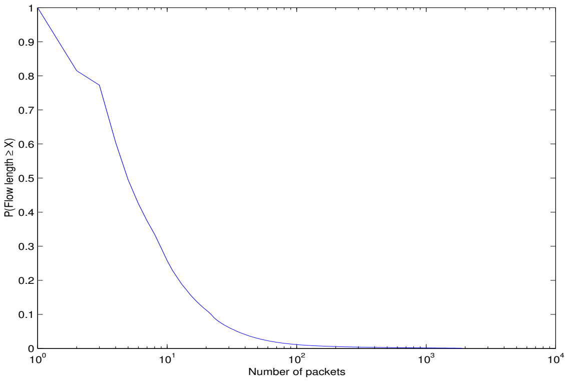


Figure 5.12: Flow length cumulative distribution.

have to be taken into account as well.

The following parameters characterize the traffic:

- *Number of packets arrived per time interval*–a discrete time homogeneous Markov chain, attaining values in the state space equal to the interval of [3000, 22000], with uniform transition probability to states within a neighboring interval of [−4000, 4000] at each iteration every 15 ms.

- *Number of flows existent in a time interval*–a stochastic recurrence, with state space attaining values in the state space equal to the interval of [8000, 240000], with uniform transition probability at each iteration every 15 ms to states within a neighboring interval that ranges between a positive and negative value of the maximum amount of packets per iteration divided by the mean number of packets per flows finished in the previous iteration. The direction of change (increase or decrease) in the number of flows is correlated with the direction of change in the number of packets (number of flows grows when number of packets grows and vice versa), as shown in [63] to hold.

- *Flow length*–the amount of packets in a flow. We have used a measured flow length distribution, as documented in [53]. The distribution is shown in Fig. 5.12. As the measured distribution covers only a finite part of the spectrum (up to flow length equal to 1886 packets), we have used Pareto distribution to approximate the heavy tail of the distribution. Other publications on the topic of flow length distribution [63] [68] [13], as well as the analysis of [2] do exhibit similar patterns and thus we believe it to be a good approximation.

- *Maximal per-flow fraction of interface rate* ϵ_f–the maximal fraction of the total rate of an interface (in packets per second) a single flow is allowed to occupy. If $\epsilon_f = 1/2$, a single flow may occupy up to 50% of the interface transport capacity. Note that the study [68] suggests that such limits on maximal per-flow rate are in conformance with reality.

- *Identifier vector values*–for the distribution of identifier vector values, we have approximated a typical distribution of IP source and destination addresses in networking traffic, as described in [45]. The prevalence of class C addresses, which occupy a relatively small portion of the address space (12.5%) and yet account for approximately 65% of the packets in network traffic, led us to consider a normal distribution of identifier vectors within a 32-bit integer space, with parameters fitted to those measured in [45]. Thus, the identifier vectors of flows are generated with a truncated normal distribution $\mathcal{N}(0, 1)$, rounded to match the entries out of the 32-bit integer space.

- *Load per packet*–we have used a simplified representation of the packet load in number of processing units it takes to process a packet. We have defined three possible levels of the per-packet load $l \in \{1, 2, 3\}$. The rationale for such a definition is an analogy with the most common router function–the address lookup–where the forwarding table is often organized into a tree structure and a lookup requires variable amount of memory accesses, depending on the tree depth per particular prefix [14] [44]. Often, the trees are organized in three levels, and thus an address lookup may require 1-3 memory accesses. In our simulations, the distribution of the per-packet load over the identifier vectors is uniform. That is certainly a simplification from reality, yet obtaining realistic values would require to match traffic traces against a correspondent lookup table, a task not within our means.

5.4 Router system model

The iterations of the traffic generation process, as well as the evaluations of the adaptation trigger, are carried out at a time interval $\Delta t = 15$ ms. Such interval should be comfortably large to be greater than any hypothetical ΔT (maximal time a packet can spend in the system), which is typically defined in the order of nanoseconds. The unit of load at each NPU is equal to 1 memory access. If not stated otherwise, we assume a homogeneous router model, where all the NPUs have equal processing capacity, corresponding to a full load of a single router interface, which amounts to 22000 packet per 15 ms, which leads to $\mu_j = 44000$ processing units (memory accesses) per 15 ms. The NPUs are considered buffer-less devices that can not process more traffic within an interval than the limit of maximum number of processing units per time interval allows. Traffic exceeding this limit per time interval is assumed to be dropped.

Where not stated otherwise, the low-pass filter constant r is set to $r = 3$ and the hysteresis bound to $\epsilon_h = 0.1$. The number of links n and processors m in the simulations have been chosen such that the total system workload intensity remains close to 1, where it makes sense to investigate the performance with respect to the acceptable load-sharing bounds.

In the subsequent simulations, three alternatives of a router system model are compared frequently, to demonstrate the functionality and advantages of the adaptive load-sharing method: a naive system where *no load sharing* is deployed over the processors and thus the entire load of any single interface is mapped to a particular processor; a *static* load-sharing system, which uses the HRW packet-to-NPU mapping f to map packets to processors, yet with mapping weights remaining static, as configured initially according to the capacities of the individual processors; and, finally, the *adaptive* system, with the dynamically adapted weights of the packet-to-NPU mapping $f_{(t)}$.

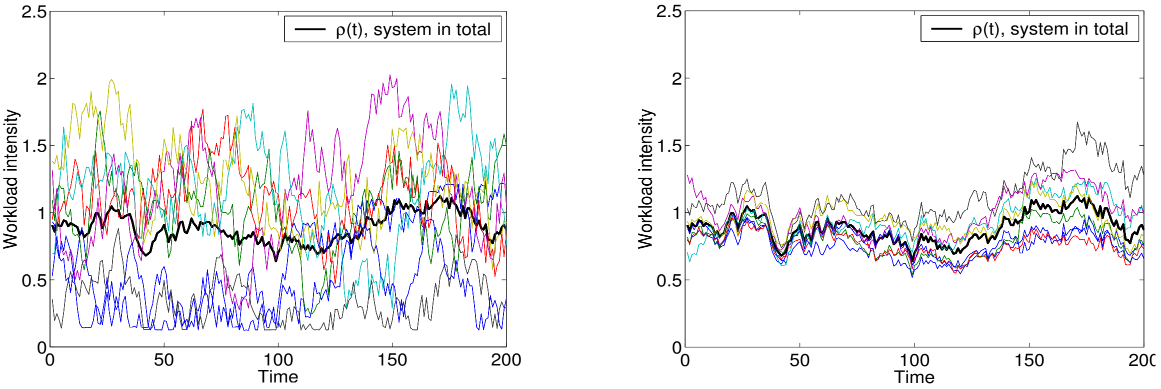
5.5 Results

5.5.1 Adaptive load sharing

Figures 5.13–5.16 illustrate on one simulation, how the adaptive load-sharing method works in general. Load of $n = 13$ links is processed by a router equipped with $m = 8$ processors. The cases where no load sharing is deployed, where load sharing is deployed using the static mapping and where load sharing is deployed using the adaptive mapping are compared. All the alternatives are put in perspective by a comparison to the ideal solution.

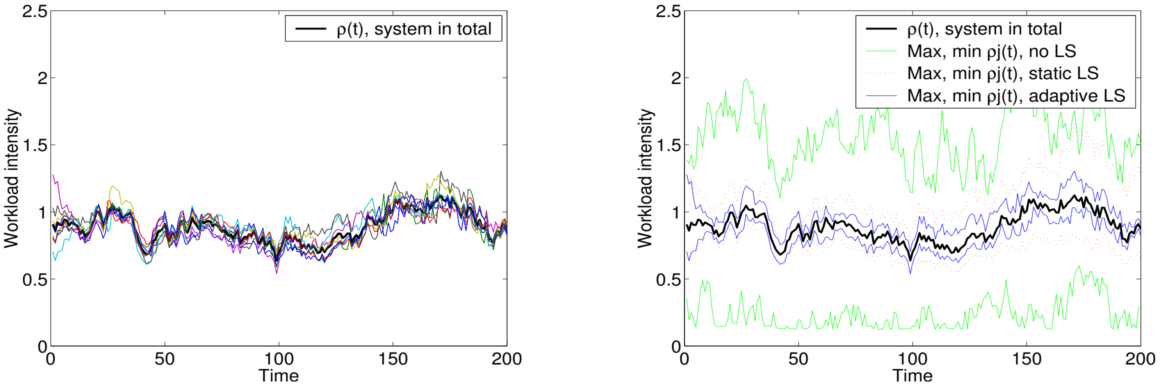
Figure 5.13 compares the per-processor workload intensity when using each method. Clearly, individual processor workload intensity remains within close vicinity of the ideal workload intensity when adaptive load sharing is deployed. Figure 5.14 compares the amount of packets dropped under the three scenarios. Again, the adaptive case results closely follow the total system curve, which represents the global minimum. Figure 5.15 depicts the functioning of the triggering and adaptation policies. In Fig. 5.16, the number of per-iteration flow remappings when using the adaptive load-sharing method is being examined. Note that the number of flows re-mapped per iteration is several orders of magnitude smaller compared to the number of flows appearing per iteration.

Relevant results of multitude of such experiments are summarized in Tables 5.1 and 5.2.



(a) No load sharing.

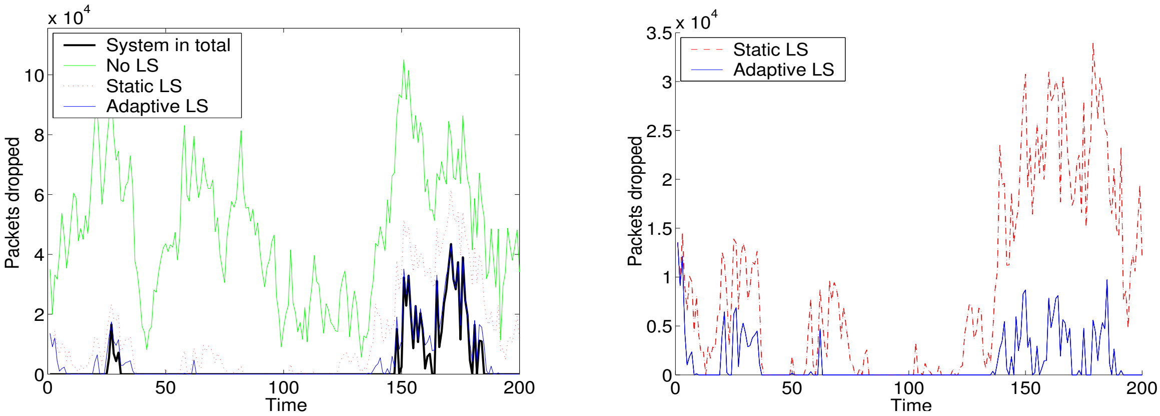
(b) Static load sharing.



(c) Adaptive load sharing.

(d) Combined.

Figure 5.13: **Per-processor workload intensity.** A load of $n = 13$ links is processed by a router equipped with $m = 8$ processors. If no load sharing is deployed, the entire load of each link is assigned to a particular processor. This is compared to a case where load sharing is deployed using a static, non-adaptive mapping f and to load sharing with the dynamically adapted mapping $f_{(t)}$. Figure 5.13(a) depicts the individual processor workload intensity when no load sharing is deployed, figure 5.13(b) the individual workload intensities when the static load-sharing method is deployed and figure 5.13(c) when the adaptive method is deployed. Figure 5.13(d) summarizes these results by showing the maximal and minimal per-processor workload intensity per each scheme. On all figures, the total system workload intensity $\rho(t)$, which is the same in all three cases, is shown for comparison, as it represents an ideal reference value. Clearly, individual processor workload intensity remains within the closest vicinity of the ideal workload intensity when adaptive load sharing is deployed.



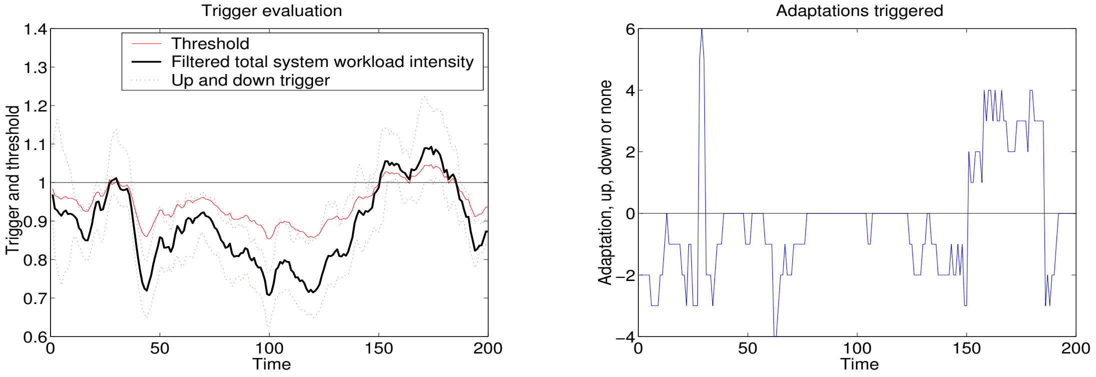
(a) Packets dropped in total.

(b) Packets dropped in excess.

Figure 5.14: **Packets dropped.** A load of $n = 13$ links is processed by a router equipped with $m = 8$ processors. The graphs depict the numbers of dropped packets when not using a load-sharing scheme, when using a static load-sharing method and when using an adaptive method. Figure 5.14(a) shows the number of packets dropped by the system using each of the schemes, compared to the ideal value of the minimal number of packets the system would have to drop, should the individual processors be ideally saturated. Figure 5.14(b) compares the static and the adaptive method in the number of packets dropped in excess of the ideal minimal value. Clearly, the adaptive method significantly outperforms the static one and avoids a large number of unnecessary packet drops, in particular during periods when the ideal solution likewise leads to no packet drops.

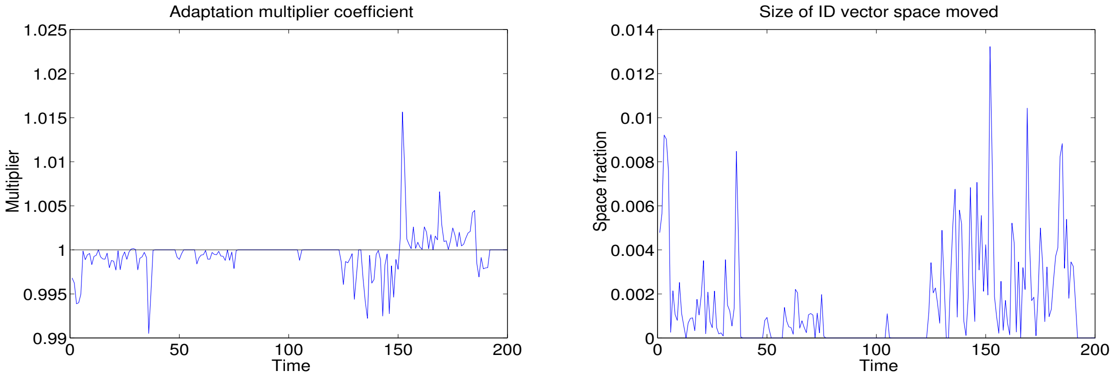
Table 5.1: Acceptable load sharing.

Packets dropped	*% of All*
Acceptable Load Sharing (Ideal)	0.7548
Adaptive Load Sharing	1.3256
Static Load Sharing	2.2590
Packets dropped in excess of Ideal	
Adaptive Load Sharing	0.5708
Static Load Sharing	1.5042
Improvement of Adaptive over Static	%
1 - (Adaptive / Static)	60.4591



(a) Trigger evaluation.

(b) Number of adaptations triggered.



(c) Adaptation coefficient.

(d) Fraction of space remapped.

Figure 5.15: **Triggering and adaptation policy.** This figure depicts the progress of the triggering and adaptation policy in the same simulation as in Fig. 5.13. Figure 5.15(a) depicts the total system workload intensity, the triggering threshold and the triggers for the upwards or downwards adaptation. Figure 5.15(b) then shows the actual amount of adaptations carried out, a positive adjustment of 1 NPU is represented by +1, a negative adjustment by -1. The adaptation coefficient used in the adaptation iterations is shown in Figure 5.15(c) and Figure 5.15(d) depicts the fraction of the identifier vector space V that changes destination in each iteration.

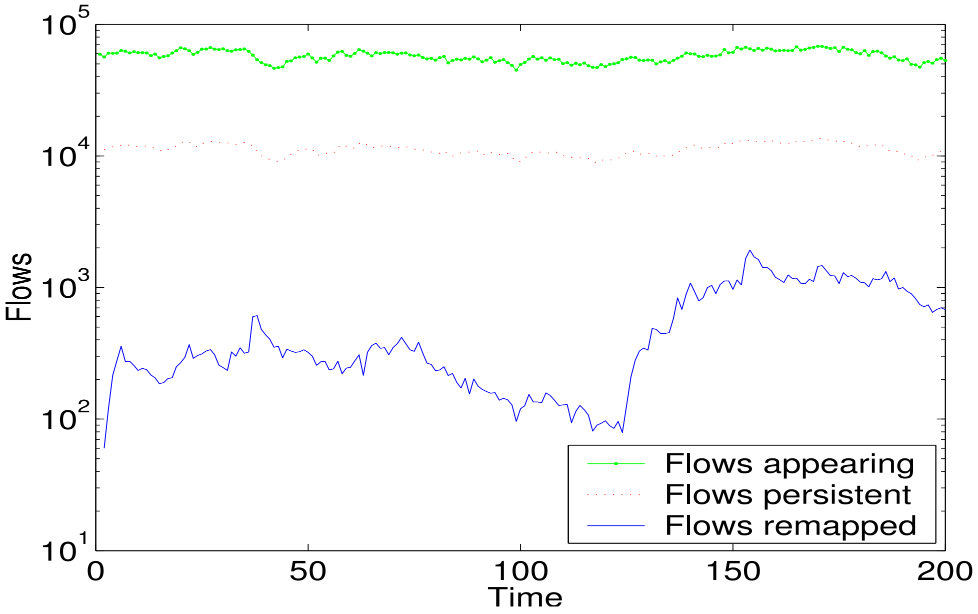


Figure 5.16: **Flows re-mapped.** Load of $n = 13$ links is processed by a router equipped with $m = 8$ processors, with the adaptive load-sharing method being executed. The figure depicts, on logarithmic scale, the statistics on the number of flow remappings occurring within an iteration. The top line shows the total number of flows appearing within an iteration, the middle one the number of persistent flows (flows that had appeared in the previous iteration as well) and the lowest one the amount of persistent flows re-mapped within an iteration. Clearly, there is at least an order of magnitude difference between each of the flow statistics, showing that only a small fraction of flows are vulnerable to remapping and out of these, only a very small fraction eventually are re-mapped.

Table 5.2: Flow remappings.

	All	Persistent	Remapped
Mean over simulations			
% of all	100.00	19.23	0.02
% of persistent	-	100.00	0.11
Per iteration			
Max, # of flows	77’076	15’304	204
Max, % of all	100.00	21.19	0.43
Max, % of pers.	-	100.00	2.22

5.5.2 Influence of maximal flow rate ϵ_f

Simulations show that the fraction ϵ_f a single flow is allowed to consume from the rate of a single interface, and, consecutively and more importantly, from the capacity of a single NPU, is a key parameter, determining the effectiveness of the adaptive method. The lower the limit on a single flow's rate, the better the method performs, as shown in fig. 5.17, where the same system model is subject to traffic varying in the maximal flow rate factor: cases of the maximum flow rate being limited to 10%, 25%, 50% of the link capacity and the case where a single flow can occupy the entire link (or NPU), are being compared. Clearly, the lower the limit, the better the adaptive method performs.

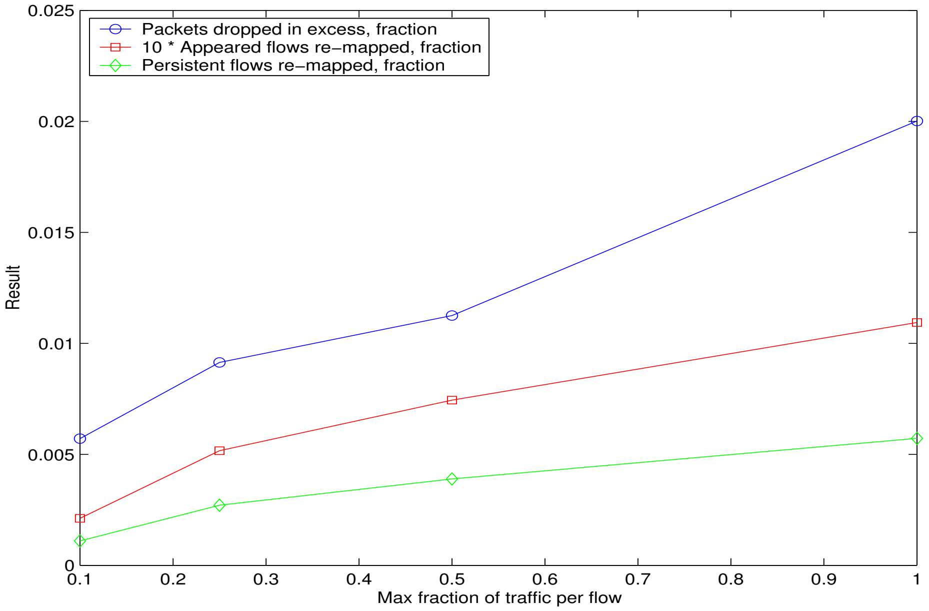


Figure 5.17: **Influence of maximal per-flow fraction rate limit ϵ_f.** The key response variables–number of packets dropped in excess, the fraction of flows re-mapped out of all flows seen per iteration and out of all flows persistent per iteration–are shown for four different settings of the maximal per-flow rate ϵ_f: 10%, 25%, 50% and 100% of the line rate, or of a single processor's capacity. All the observed response variables grow with the increasing ϵ_f, which thus has a negative impact on the method's performance.

5.5.3 Fractional factorial analysis of the load-sharing method

To test the influence of various tunable parameters (factors) on the adaptive method's performance, we have employed the technique known as Fractional Factorial Analysis [28]. Upon alternating the values of each factor, one can measure the influence of each factor on the variance of the results.

Each factor has been alternated between two values (levels)–high (Hi) and low (Lo). The following factors and levels have been explored:

- *Adaptation coefficient exponent.* The levels used were $Lo = 1/\log_2(m)$ and $Hi = 1/m$;

- *NPU processing capacity distribution.* For level Lo, all the individual capacities were equal, whereas level Hi implied an exponential distribution of capacities, with relative capacities in $\{1, 2, 4, 8\}$;

- *Adaptation interval.* The levels used were $Lo = 1$, meaning the adaptation would be carried out at every iteration, if triggered, and $Hi = 10$, meaning the adaptation would only be carried out every 10th iteration;

- *Hysteresis bound ϵ_h.* The levels used were $Lo = 0.01$ and $Hi = 0.1$;

Table 5.3: Fractional factorial analysis results.

No.	Exponent	NPU capacity	Adaptation interval	Hysteresis	Filter	*Packets dropped*	*Flows re-mapped*
1.	-1	-1	-1	1	-1	87091	700
2.	-1	-1	1	-1	1	223181	422
3.	-1	1	-1	1	1	203333	306
4.	-1	1	1	-1	-1	164660	438
5.	1	-1	-1	-1	1	177332	858
6.	1	-1	1	1	-1	228731	190
7.	1	1	-1	-1	-1	119675	926
8.	1	1	1	1	1	266604	16
Total / 8						183830	482

- *Filtering parameter r.* The levels used were $Lo = 3$ (meaning significant influence is given to the recently measured value) and $Hi = 32$ (meaning the recently measured value has less influence).

The results of the experiments are shown in Table 5.3, where -1 represents the Lo level and 1 the Hi level. The results on the impact of each factor on the variance of the key response variables, the number of *packets dropped in excess* and the number of *flows re-mapped* are shown in table 5.4.

The Fractional Factorial Analysis results show that out of the factors studied, only three have significant influence on the results: the adaptation interval, the hysteresis bound and the filtering parameter.

The most significant factor appears to be, not surprisingly, the *adaptation interval.* Clearly, more frequent adaptation leads to more accurate load sharing and thus less packet loss, yet more frequent adaptation leads to more flow remappings as well. The exact duration of the adaptation interval thus needs to be set according to the desired parameters of the system.

The amount of packets dropped is likewise significantly influenced by the *filtering parameter r.* As this factor, in comparison, has relatively little influence over the amount of flows re-mapped, it makes sense to choose a value of r which leads to less packet drops. Clearly, when r is high, the system does not take into account the recent information strongly enough. A lower value, like the $r = 3$ in the experiment, can thus be recommended.

Table 5.4: Effects of factors.

Factor	*Packets dropped*		*Flows re-mapped*	
	Estimate	% of Variation	Estimate	% of Variation
Total / 8	183830		482.0	
Adaptation exponent	14260	0.066	15.5	0.003
NPU capacity dist.	4742	0.007	-60.5	0.040
Adaptation interval	36968	0.443	-215.5	0.512
Hysteresis bound ϵ_h	12164	0.052	-179.0	0.354
Filter parameter r	33787	0.370	-81.5	0.073

The second most influential factor on the variance of the number of flow remappings is the *hysteresis bound* ϵ_h. The hysteresis bound, on the contrary, has little effect on the amount of packets dropped and thus clearly it makes sense to set the bound to a larger value, for example the $\epsilon_h = 0.1$ used in the experiment, in order to prevent unnecessary flow remappings.

The relatively insignificant influence of the NPU processing capacity distribution factor confirms that the HRW mapping and the weights' adaptation work equally well with both uniform and highly non-uniform distributions of processing capacities. Likewise, the exact value of the adaptation coefficient exponent does not have high influence. Such outcome thus favors using the $\log_2(m)$ option, as computing the root in Eq. (4.26) is then less demanding.

5.5.4 Influence of the number of processors

In this simulation we evaluate the effects of alternating the total number of processors present in the system, m. We compare systems where the total processing capacity remains constant, yet the number of NPUs present and their processing capacity differs. The scenarios compared involve again a router with $n = 13$ interfaces and $m = 8$ NPUs of uniform processing capacity μ_j, plus a system with $m = 64$ processors, where each NPU has processing capacity $\mu_j/8$.

At the same time we compare the effects of alternating the exponent in the adaptation coefficient between the values of $1/m$ and $1/\log_2(m)$, to observe which of the two more accurately reflects the higher number of processors present. Furthermore, we alter the maximal per-flow fraction ϵ_f, as the maximal fraction a single flow is allowed to consume from a single interface's rate is reflected in the maximal fraction of processing capacity a single flow may consume at a single NPU. Thus, in one set of simulations when $m = 64$, we decrease ϵ_f according to the ratio of the decrease of a single NPU's processing capacity, 1/8. Thus, ϵ_f attains a value of either $\epsilon_f = 0.1$ or $\epsilon_f = 0.0125$.

The individual experiments and the results are summarized in Table 5.5 and Fig. 5.18. Clearly, as shown in Section 5.5.2, the influence of the maximal per-flow rate ϵ_f

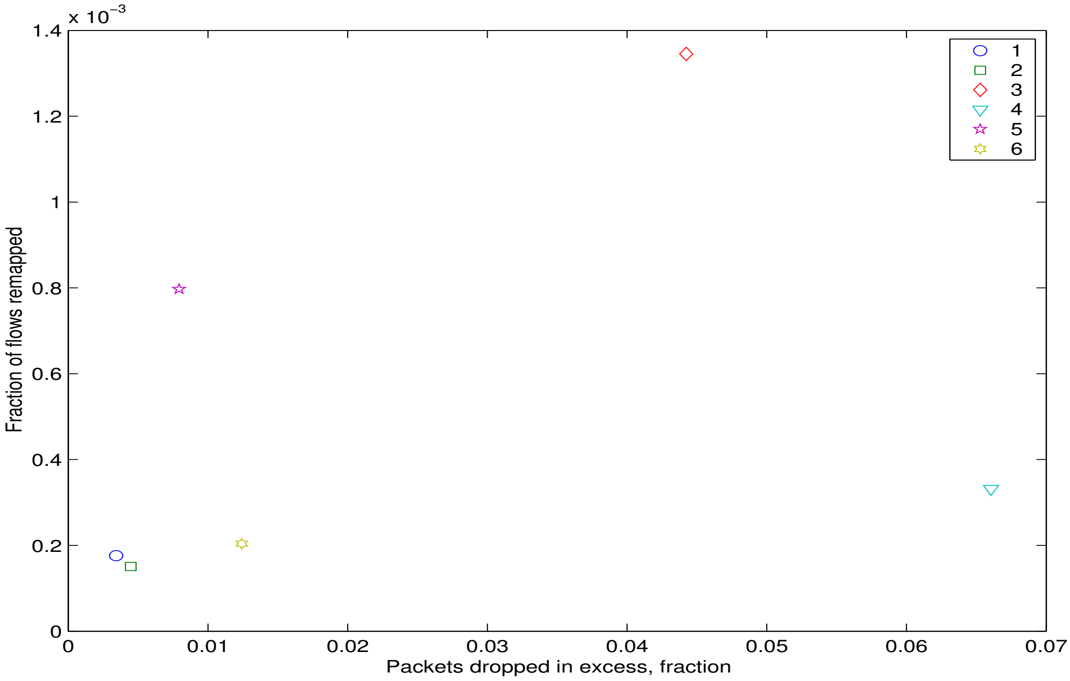


Figure 5.18: **Influence of the number of NPUs.** This figure depicts graphically the results presented in Table 5.5. Experiments 1 and 2 are conducted with $m = 8$ processors and 3-6 with $m = 64$ processors. In 5 and 6, the maximal per-flow rate fraction ϵ_f is adjusted accordingly and thus results in these two experiments do not differ too significantly from results of 1 and 2. However, clearly, with higher number of processors, the accuracy of the method is affected.

is key for the results of experiments No. 3 and 4 and thus in order to fairly evaluate the effects of the number of processors factor m, the maximal per-flow rate ϵ_f needs to be adjusted accordingly, as in experiments No. 5 and 6.

When ϵ_f is adjusted, results of the system with $m = 64$ NPUs (No. 5 and 6) differ not as significantly from results on a system with $m = 8$ NPUs, although the performance does worsen in both of the key response variables. The effects of the alternative exponents in computing the adaptation coefficients can be compared. Using $1/m$ leads to less aggressive adaptation of the mapping weights, and thus fewer flows are re-mapped. In contrast, the less fine-grained $1/\log_2(m)$ adjusts the mapping more aggressively and thus prevents packet loss, but at a cost of more flow remappings.

Table 5.5: Influence of the number of NPUs.

No.	m	Adaptation exponent	Maximal flow rate (%)	Packets dropped in excess (%)	Flows re-mapped (%)
1	8	$1/\log_2(m)$	10.00	0.343	0.0176
2	8	$1/m$	10.00	0.447	0.0151
3	64	$1/\log_2(m)$	10.00	4.423	0.1345
4	64	$1/m$	10.00	6.605	0.0332
5	64	$1/\log_2(m)$	1.25	0.795	0.0797
6	64	$1/m$	1.25	1.242	0.0204

5.6 Conclusions

In this chapter, the proposed adaptive load-sharing method has been validated. First, the basic concept was verified on a simple model with some naive generated input traffic. Then, more sophisticated modelling techniques have been used to evaluate the influence of various system parameters (factors) on the method performance when using realistically generated traffic as input.

We conclude that the theoretically derived results from Chapter 4 have been verified by the simulations. At a modest rate of remapping on average less than 0.1% of all flows, on average 60% of packets dropped by the static scheme are protected by the feedback-based adaptation.

Chapter 6

Applications in networking systems

6.1 Introduction

In this chapter, we explore two of the possible applications of the adaptive load-sharing method in multiprocessor network nodes.

In the first section, we present its applicability when designing *multiprotocol routers*. Examples of router architectures which benefit from deploying the method are given and some of the implementation issues are discussed, especially the pseudo-random function implementation.

In the second part of the chapter, we discuss the application of the method in a *server farm load balancer* appliance. We show how the method is extended to avoid flow remappings altogether. A description of an existing implementation on the IBM PowerNP network processor is given, together with a discussion of some of the implementation issues, like the data structure used for storing the HRW mapping weights.

6.2 Multiprotocol router

6.2.1 Router architecture

An implementation of a router combining the distributed router architecture with the load-sharing scenario is depicted in Figure 6.1. In this distributed router system, an NPU resides at every line card and the NPUs are interconnected through a switch. The NPUs use the switch for switching data packets as well as for exchanging the load among themselves.

The processors do not need to be dimensioned to handle the full link capacity—in case of overload, the other processors will be able to help. Likewise, the system fault tolerance is enhanced—in case of an NPU failure, the load of the interface affected can be handled seamlessly by the other NPUs within the system. Depending on the switch round-trip time, the mapping weights may be adjusted to give preference to local processors.

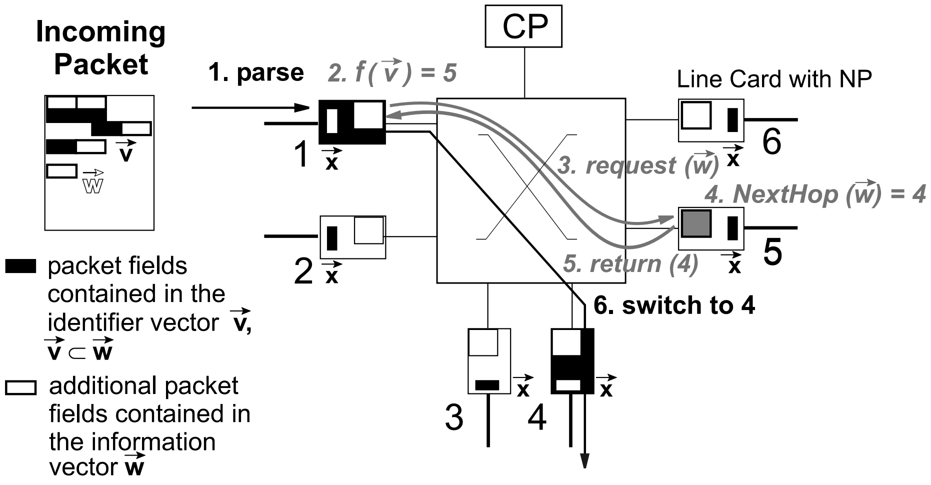

Incoming Packet
$\vec{v}$
$\vec{w}$
■ packet fields contained in the identifier vector $\vec{v}$, $\vec{v} \subset \vec{w}$
▭ additional packet fields contained in the information vector $\vec{w}$
1. parse
2. $f(\vec{v}) = 5$
CP
Line Card with NP
1 $\vec{x}$
2 $\vec{x}$
3. request $(\vec{w})$ $\vec{x}$
6
4. NextHop $(\vec{w}) = 4$
5. return (4) $\vec{x}$
5
6. switch to 4
$\vec{x}$
$\vec{x}$
3
4

Figure 6.1: **Load sharing within a distributed multiprotocol router.** Upon packet arrival, the packet information vector $\vec{w}$ travels through the switch to NPU j, chosen by computing the mapping $f(\vec{v}) = j$ over the identifier vector $\vec{v}$. At NPU j, the vector $\vec{w}$ is processed and the resolution information is returned back to the requesting NPU. The packet is then switched to the correct output port and the corresponding packet alterations or manipulations, based on the resolution results, are applied.

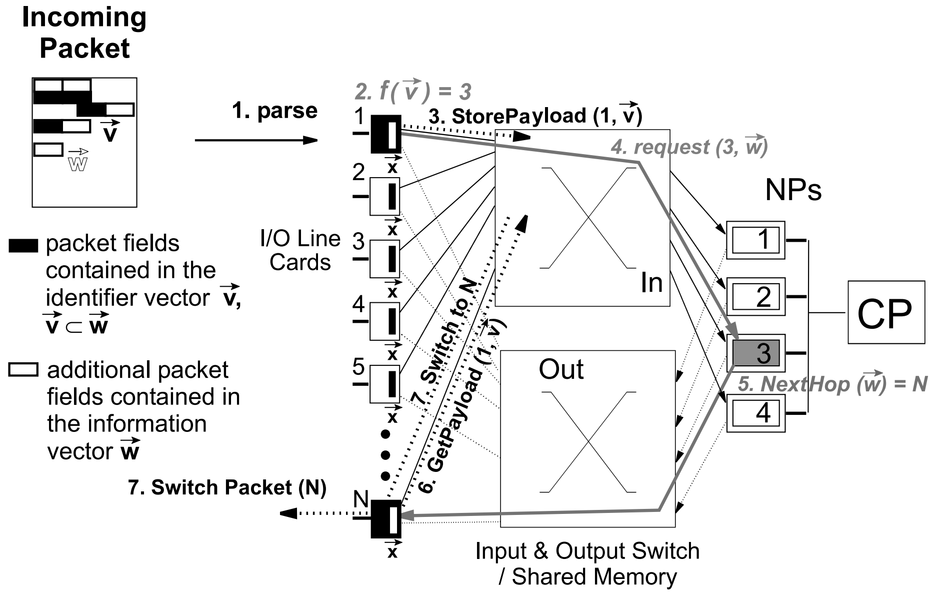

Incoming Packet
$\vec{v}$
$\vec{w}$
1. parse
2. f($\vec{v}$) = 3
3. StorePayload (1, $\vec{v}$)
4. request (3, $\vec{w}$)
5. NextHop ($\vec{w}$) = N
6. GetPayload (1, $\vec{v}$)
7. Switch to N
7. Switch Packet (N)
■ packet fields contained in the identifier vector $\vec{v}$, $\vec{v} \subset \vec{w}$
▭ additional packet fields contained in the information vector $\vec{w}$
I/O Line Cards
1
2
3
4
5
N
$\vec{x}$
In
Out
NPs
1
2
3
4
CP
Input & Output Switch / Shared Memory

Figure 6.2: **Multiprotocol router, consisting of an input- and output switch/shared memory and a pool of multiple NPUs, sharing the load of N line cards.** Every line card is connected to two switches/shared memories, input and output. Each of the multiple NPUs is reachable from a line card through a switch. Upon packet arrival, the packet is buffered in the input shared memory and the information vector $\vec{w}$ travels through the input switch to NPU j, chosen by computing the mapping $f(\vec{v}) = j$ over the identifier vector $\vec{v}$ at the line card. At NPU j, the vector $\vec{w}$ is processed and the resolution information is then switched to the resulting port of the output switch. The packet is then retrieved from the shared memory, the corresponding packet alterations or manipulations, based on the resolution results, are applied and the packet is transmitted on the link.

Another potential implementation is shown in Figure 6.2. It is a load-sharing extension of the concepts of the Juniper Networks routers, where the header and the payload-processing paths are separated by two switches, input and output. The control information in the packet header is processed in a remote NPU, while the payload is temporarily stored in a distributed shared memory coupled to the input switch. Sharing the load among the NPUs brings significant utilization benefits. Furthermore, in this manner a centralized load balancer over the pool of NPUs is not needed and thus a potential single point of failure within a router is being eliminated.

6.2.2 Load indicator

An open implementation issue is how to actually measure the load of the processors or the amount of processing units spent per time interval. Alternative measures to the one used in the simulations (number of memory accesses an NPU has performed during the time interval Δt) can be for example the number of processing cycles or the number of packets processed per time interval. A counter value is periodically read by the CP. Multiple indicators can be combined into one using relative weights, as in the Network Dispatcher [24] application (see Section 2.4.3, yet note that results of Kunz [35] indicate that the benefit over a one-dimensional load descriptor is minimal.

Although the load information gathering appears to present unnecessary extra effort for the router system, note that similar statistics collection is readily available and often required from the existing router equipment [18] [38] and therefore should result in minimal load increase on the system.

6.2.3 Pseudo-random function

The major implementation issue related to the load-sharing method is how to provide a fast computable pseudo-random function g for computing the HRW mapping f, with the properties required in the mapping definition (Def. 3).

The device performing the mapping must compute N uniformly distributed pseudo-random values (let us call them $score_i$) using some packet fields ($vecv$) and the outgoing interface index i. These values are later used as inputs for the mapping function that selects the forwarding port.

Let us recall some desired properties of the function:

- The function must map (at least) a 32-bit input to a 32-bit output score. Some flow-dependent packet fields, such as the source IP address or the source port number are used as input.

- The function should generate uniformly distributed and uncorrelated scores. The number of collisions (different inputs that generate the same output score) should be minimal.

- The function must be deterministic–it must generate the same score for all the packets of the same flow. This avoids that packets of the same connection are being forwarded to different processors.
- We can not assume that the inputs are uniformly distributed. Nevertheless, the output of the function should not show the effects of the bit correlation of the input.

We propose a list of hash functions that scramble and mix the input digits to obtain a score. We discuss their usability and evaluate the spreading quality and the number of operations involved in the hashing process.

Knuth's standard integer hash function

Knuth [32] recommends the following integer hash function:

```
for (hash=len; len--;)
 hash = ((hash<<5)^(hash>>27))^*key++;

hash = hash % table_size;
```

Unfortunately, this hash function shows only mediocre scrambling performance. The problem is the per-character mixing: it only rotates bits, instead of mixing them. It can be demonstrated that every input bit affects only 1 bit of the hash.

Fibonacci golden ratio multiplicative hash function

In Section 6.4 of [32], Knuth introduces a multiplicative hashing scheme, which seems to be a good candidate for the mapping purposes. It is based on the Fibonacci golden ratio multiplier $\phi^{-1} = (\sqrt{5} - 1)/2$. The Fibonacci hash function leads to the "most random" scrambling of sequences [32]. It is defined as follows:

$$h_{\phi^{-1}}(x) = (\phi^{-1}\, x) \bmod 1. \tag{6.1}$$

In a 32-bit arithmetic, the key is multiplied by the golden ratio ϕ^{-1}, computed over 32 bits (2654435769) to produce a hash result:

```
unsigned int inthash(unsigned int  key) {
    return (key*2654435769) & 0xFFFFFFFF;
}
```

Because 2654435769 and 2^{32} have no common factors, the multiplication produces a complete mapping of the key to the hash result with no overlap. This method works nicely for keys with small values. The spread of hash results is not so good if the keys vary in the upper bits. As is true for all multiplications, variations of upper digits do not influence the lower digits of the multiplication result. Furthermore, this method requires a multiplication by a fixed number, which may be a cycle-intensive operation for some systems.

The function can be fit into the mapping scheme as follows:

$$g(\vec{v}, j) = h_{\phi^{-1}}(\vec{v} \;\text{ XOR }\; h_{\phi^{-1}}(j)). \tag{6.2}$$

As the values $h_{\phi^{-1}}(j)$ can be pre-computed, the actual computation per vector $\vec{v}$ requires only $4m$ basic operations and m comparisons (to find the maximum).

In our simulations in Chapter 5, we have used Fibonacci hashing to compute $g(\vec{v}, j)$.

Jenkins and Wang's 32-bit Mix function

As multiplication is not always straightforward to implement, a fast computable function consisting of only simple bit operations may be necessary. Jenkins [29] defines the 96-bit Mix function:

```
#define mix(a,b,c) \
{ \
  a=a-b;  a=a-c;  a=a^(c>>13); \
  b=b-c;  b=b-a;  b=b^(a<<8);  \
    c=c-a;  c=c-b;  c=c^(b>>13); \
    a=a-b;  a=a-c;  a=a^(c>>12); \
    b=b-c;  b=b-a;  b=b^(a<<16); \
    c=c-a;  c=c-b;  c=c^(b>>5);  \
    a=a-b;  a=a-c;  a=a^(c>>3);  \
    b=b-c;  b=b-a;  b=b^(a<<10); \
    c=c-a;  c=c-b;  c=c^(b>>15); \
  }
```

The input parameter `c` initially contains the input key and `a` and `b` are set by default to the 32-bit value of the golden ratio ($\phi^{-1} = (\sqrt{5} - 1)/2$). At the end of the computation, the hash result is stored in `c`.

From this function, Jenkins derives a 32-bit Mix Function:

```
unsigned int inthash(unsigned int key) {
  key += (key << 12);
  key ^= (key >> 22);
  key += (key << 4);
  key ^= (key >> 9);
  key += (key << 10);
  key ^= (key >> 2);
  key += (key << 7);
  key ^= (key >> 12);
  return key;
}
```

Wang [66] proposes a faster computable version of the above:

```
unsigned int inthash(unsigned int  key) {
  key += ~(key << 15);
  key ^=  (key >>> 10);
  key +=  (key << 3);
  key ^=  (key >>> 6);
  key += ~(key << 11);
  key ^=  (key >>> 16);
  return key;
}
```

Depending on the computing capacity available, other hash functions may be used. The study in [7] documented good spreading properties of the CRC16 function, yet at a cost of higher complexity.

6.3 Server load balancer on a network processor

This describes a method where multiple servers forming a server farm are connected to a front-end Network Processor, which balances the processing load among the servers. We present an existing implementation of the method on the IBM PowerNP network processor.

6.3.1 Server farm topology

The NP acts as a Virtual Server, representing the N real servers belonging to the Server Farm and connected to the NP. The IP address of the Virtual Server is the IP address of the entire farm to the outside world. When a packet arrives, the NP checks whether the destination IP address is the server farm address. If *not*, it proceeds with the standard IP forwarding; *otherwise* it starts the specific forwarding algorithm.

The server farm must be transparent to the Internet users. They must perceive the connection as if connected to a Virtual Server, impersonated by the NP, even if the NP switches the flow to a specific real server according to the load balancing policy. This means for example that all the packets received by the user must have the IP address of the Virtual Server. The are two different approaches to achieve this:

1. The real servers all have a private IP addresses, and the NP implements the NAT function. This solution hurts the NP performance because the NP must:

   - change the IP destination address of all incoming packets to the private IP address of the real server to which the packets will be forwarded;
   - change the IP source address of all outgoing packets to the Virtual Server IP address.

   Note that NAT translation involves resource-intensive operations, such as re-computing the spacket checksum;

2. All real servers have the same IP address (identical to the address of the Virtual Server). The real-to-virtual server connection must be a point-to-point connection. Such a solution may lead to problems in the ARP/RARP protocol operation, because the NP is linked to several nodes and shares the same IP address with them. To avoid these problems, we have disabled the ARP/RARP protocol on the interfaces towards the real servers and put some fixed entries directly in the NP ARP tables for the IP/MAC mappings of the real server.

For our implementation, we have chosen the second alternative, for its speed and ease of management. Fig. 6.3 shows the network topology used during testing. There are three real servers and two hosts, acting as Internet users. Tables 6.1 and 6.2 show the corresponding IP routing table and the ARP table, as stored in the Linux Control Point of the NP:

Entries on lines ‘B’, ‘C’, ‘D’, ‘E’, ‘F’ and ‘G’ are pre-configured. Note that line ‘F’ is the link to the Control Point. Line ‘P’ is the default gateway. Thanks to lines

Table 6.1: Kernel IP routing table

Line	Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
A	12.0.0.2	*	255.255.255.255	UH	0	0	0	reth2
B	12.0.0.0	*	255.255.255.0	U	0	0	0	reth2
C	12.0.0.0	*	255.255.255.0	U	0	0	0	reth3
D	12.0.0.0	*	255.255.255.0	U	0	0	0	reth4
E	31.0.0.0	*	255.255.255.0	U	0	0	0	reth21
F	3.3.3.0	*	255.255.255.0	U	0	0	0	npctl0
G	15.0.0.0	*	255.0.0.0	U	0	0	0	reth5
H	127.0.0.0	*	255.0.0.0	U	0	0	0	lo
I	default	k64route.zurich	0.0.0.0	UG	0	0	0	eth0

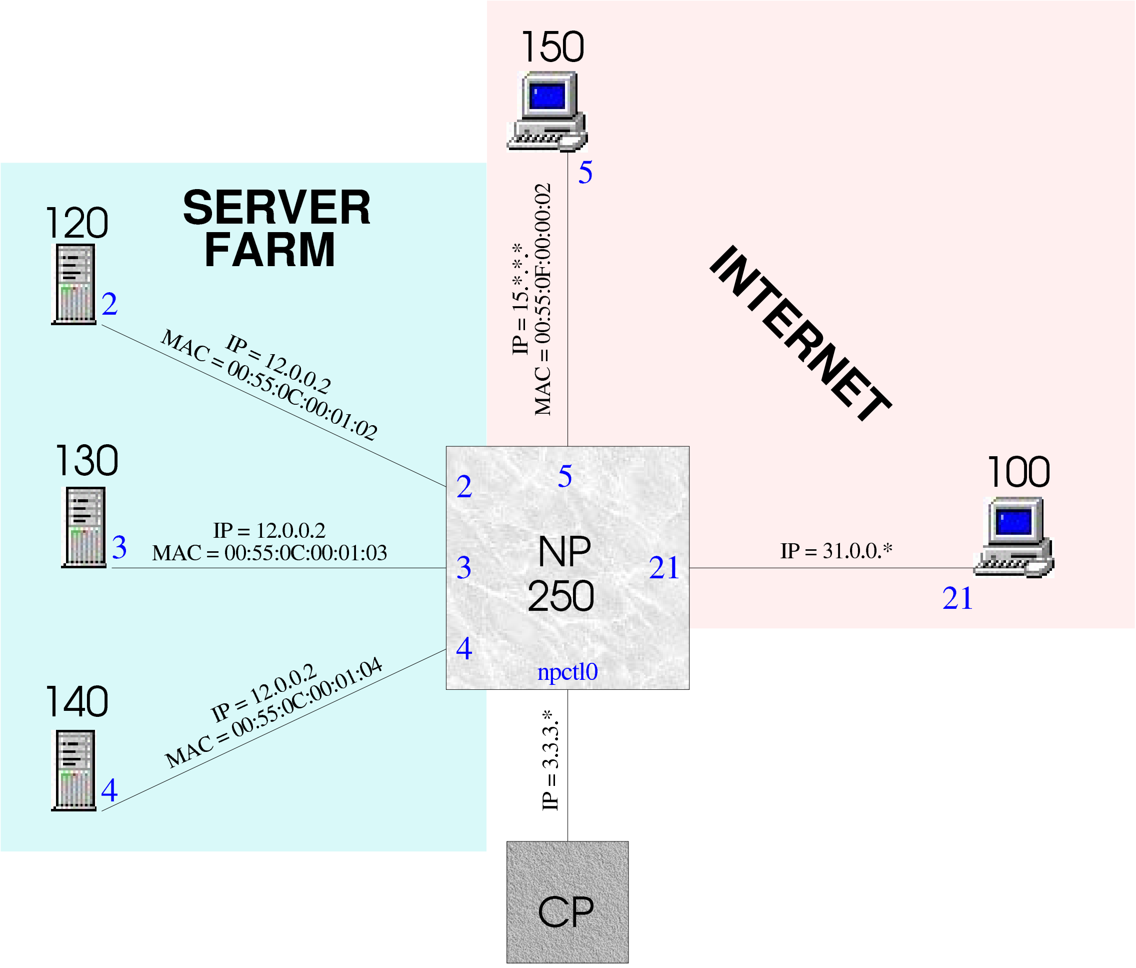


Figure 6.3: Sample server farm topology

‘L’, ‘M’ and ‘N’, the NP does not have to send an ARP request when it tries to forward a packet (with destination IP equal to 12.0.0.2) to a real server, which is on the interface reth2, reth3 or reth4, because it already knows the destination MAC address of the network card of that machine.

Table 6.2: Kernel ARP table

Line	Address	HWtype	HWaddress	Flags Mask	Iface
L	12.0.0.2	ether	00:55:0C:00:01:02	CM	reth2
M	12.0.0.2	ether	00:55:0C:00:01:03	CM	reth3
N	12.0.0.2	ether	00:55:0C:00:01:04	CM	reth4
O	15.0.0.2	ether	00:55:0F:00:00:02	CM	reth5
P	k64route.zurich.ibm.com	ether	00:00:0C:07:AC:00	C	eth0

6.3.2 Load Balancer on the PowerNP network processor––general overview

The task of carefully balancing the load among multiple servers is non-trivial due to the high volume (and thus high speed requirements) of data traffic and a contradictory requirement of preserving the assignment of flows to servers, once established. To best balance simultaneously a large number (10,000s or 100,000s) of packet flows in hardware, it is necessary to determine a scheme that keeps minimum amount of state on flows, but maintains the connectivity of active TCP flows between hosts. For TCP flows, we must guarantee that the two end-points (the user and a Real Server) do not change until the connection is explicitly ended (i.e. the Real Server receives a FIN or RST message). As for the UDP packets, as the UDP protocol is connectionless, we can forward the packets to different Real Servers without problems (considering no fragmentation).

Zero flow remappings

When the NP receives a packet for the server farm, it computes the outgoing interface (Target Port) using the robust hash routing mapping algorithm. At the initial phase, one mapping function is configured, with the weights based on the resource capacities of the servers. If the mapping weights do not change, the forwarding algorithm assures *flow preservation*, meaning that all packets of the same flow are forwarded to the same Real Server. The deterministic hash computation is performed on a portion of the packet that remains constant for the duration of a flow, such as the IP source address and the source port.

Once the method has been put into operation, statistics are accumulated on the resources utilization. If some servers become over-utilized relative to other servers, a new mapping computation is determined. The mapping computation H_{new} optimally distributes network traffic based on the statistics gathered.

Assume that at time $t = t_0$ the CP adjusts the weights (using the adaptation policy) and thus there may exist a TCP connection, whose packets, that arrive at the NP at time $t > t_0$, are forwarded to a new Real Server, which cannot handle such

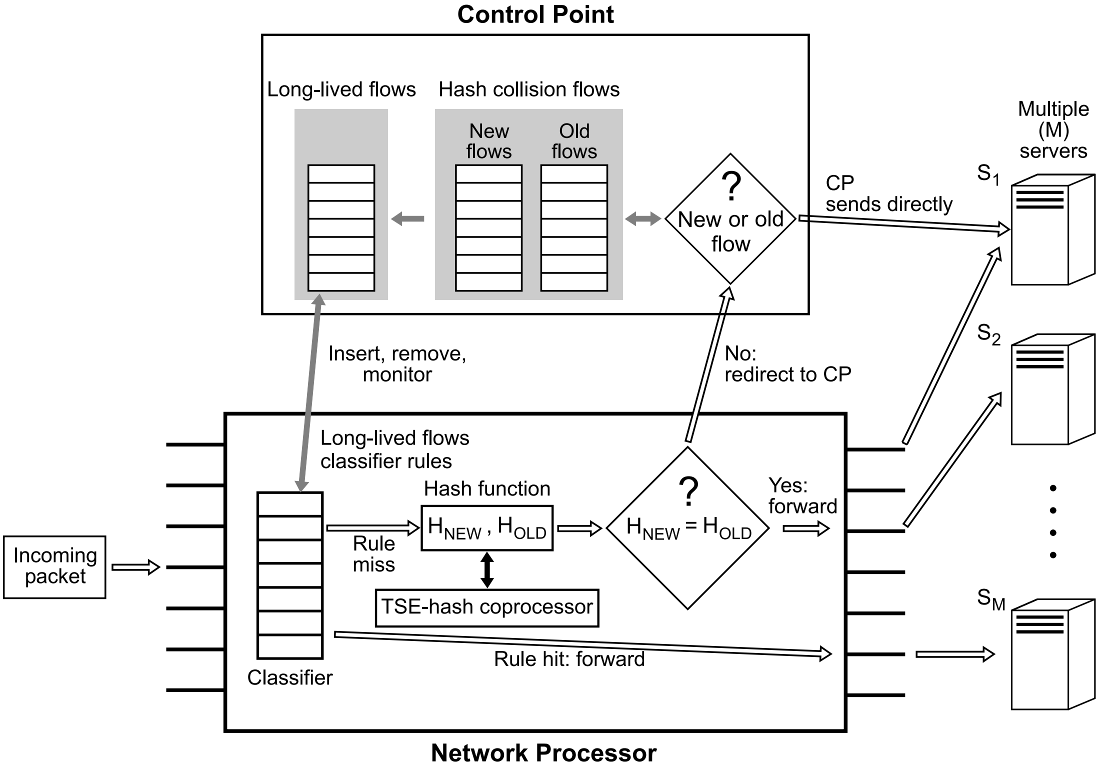


Figure 6.4: Server load balancer on the PowerNP network processor.

previously existing TCP connection, as it does not have any buffered state for it.

The solution is to alternate between a state where only one mapping is computed, and a transient state, where two mappings are computed. During the transient period, the method is complemented by a small set of Classifier rules that do treat the minimum amount of flows affected by the transient state.

During the transient period, both the old (H_{old}) and the new (H_{new}) mappings are computed on each packet simultaneously. Packets in the intersection of the two mappings ($H_{new} = H_{old}$) continue to be routed to the resulting server. Packets that do not fall in the intersection of the two mappings ($H_{new} \neq H_{old}$) are redirected to the CP for routing. The CP keeps state for each flow it sees and routes flows in progress (flows where no SYN bit is seen) to the result of the old hash function and new flows (flows where the SYN bit is seen) to the result of the new hash function. The state for flows that are finished (flows where the FIN bit is seen) or do time-out is deleted.

After a configured period of time, Multi-Field Classification rules, which specify to which server the flow is routed, are installed for the remaining (”long-lived”) flows in the state table. The old mapping function is then discarded and the method returns to a state of a single mapping. The Multi-Field Classification rules are being further monitored and removed upon flow termination or time-out.

The method continually alternates between the one-mapping and two-mapping states, to continually optimize the load balance for current traffic conditions.

Advantages of this approach include:

- No flows in progress are ever moved between servers, ensuring uninterrupted flow connectivity;
- State information is only maintained for flows not in the intersection of the two mapping functions, minimizing hardware costs;
- The intersection of the two mapping functions is mathematically maximized, thus further minimizing the state kept.
- Routing is performed partially in hardware, using hashes performed by the TSE coprocessor in the NP, thus exploiting the high data rate of the device.

6.3.3 Network processor data plane

The NP must first determine whether the packet is classified as belonging to the set of long-living flows. For all these packets, regular IP forwarding is disabled, and the NP uses the information provided by the classifier rule for forwarding the packet.

The forwarding algorithm, based on the robust hash routing, is only carried out (using the sets of the new and the old weights) on the non-filtered packets, destined to the Server Farm. Using TP_{New} and TP_{Old}, the NP forwards the packet to TP_{New} in case of equality or to CP in case of ambiguity. Non-filtered packets having a destination IP that is not equal to the Server-Farm IP are routed normally: the NP uses the routing tables to determine the Target Port.

Fig. 6.5 shows the state diagram for the forwarding algorithm carried out on the data plane of the NP:

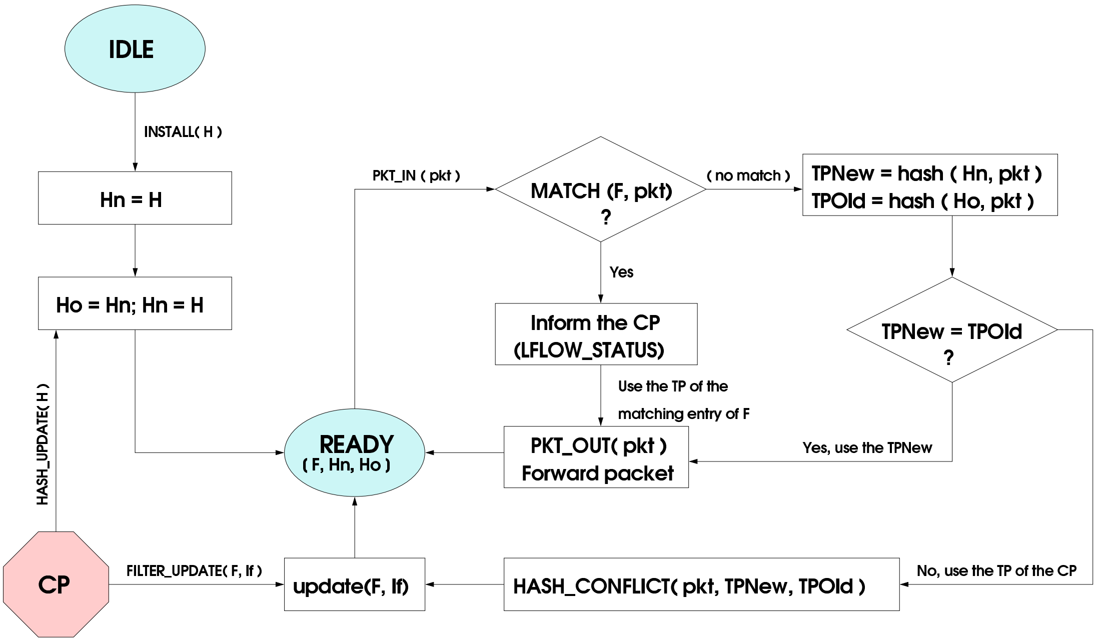


Figure 6.5: NP state diagram

INSTALL (H) is an API that, considering the percentages set by the user, installs the corresponding set of weights (H) in the NP memory;

HASH_UPDATE (H) is a similar API that modifies the set of weights. This is done by overwriting the values of the old set (H_{old}) with those of the new set and installing in H_{new} the set passed as parameter (H);

PKT_IN (pkt) simulates a new packet destined for the Server Farm, arriving at the NP;

MATCH is performed by the Multi-Field Classifier which can filter and handle the packets (LFLOW_STATUS function) based on the filtering rules (F) installed by the CP (FILTER_UPDATE);

HASH_CONFLICT is the redirection of the packet to the CP. Note also that together with the packet also the TP_{new} and TP_{old} must be sent.

PKT_POUT (pkt) is the setting of the TP and consequently the eventual forwarding of the packet.

6.3.4 HRW weights representation on the network processor

The robust hash routing algorithm requires that the probability distribution of the hashed values is $U(0,1)$. These uniformly distributed values then have to be multiplied by the weights.

This is complicated to execute for several reasons. One is that the PowerNP assembler-based pico-code does not include a specific operand for the multiplication. If we need to multiply two 16-bit registers, it is quite easy to see that the time complexity is $\sim O(\text{const} \cdot 16)$. Moreover the multiplication must be executed for every packet and for all N ports. Thus, the time complexity for the overall forwarding process presented in (2.3) is $\sim O(\text{const} \cdot 16 \cdot N)$.

The following formula proposes a new way to compute the weights x_i. The weights are now used as additive offsets. The idea is that by applying the logarithm, which is a monotonous increasing function, to both sides of Eq. (2.3), we preserve the equality:

$$
\begin{aligned}
f(\vec{v}) &= j \\
&\Longleftrightarrow \\
\log(x_j \cdot h(\vec{v},j)) &= \max_{k\ \in[1,N]} \log(x_k \cdot h(\vec{v},k))
\end{aligned}
\tag{6.3}
$$

The logarithm of a product can be split into the sum of the logarithms of the two numbers and thus we do not have to compute a multiplication, but a logarithm. Recalling the inverse transform method for the continuous distributions, we note that the logarithm of uniformly distributed values produces an exponential distribution (in our case a negative exponential). The following lemma proves that the key properties of the HRW mapping hold for after the logarithm substitution as well:

Lemma 2 (R. Russo) *Let* $p_1, \ \ldots, \ p_N$ *be given target probabilities. Reorder the targets so that* $p_1 \leq \ \ldots \leq p_N$*. Let*

$$x_1 = \frac{\ln N p_1 + S}{N},$$

with $S = \sum_{i=1}^{N} x_i$, *and let* $x_2, \ldots, x_N$ *be calculated recursively as follows:*

$$x_n = \frac{1}{N-n+1} \cdot \ln \left\{ \frac{(p_n - p_{n-1})(N-n+1)}{\exp\left(\sum_{i=1}^{n-1} x_i - S\right)} + \exp\Big(x_{n-1} \cdot (N-n+1)\Big) \right\} \tag{6.4}$$

Then the robust hash routing algorithm with offsets $x_1, \ldots, x_N$ *for every score* $h_1, \ldots, h_N$ *will route the fraction* p_n *of incoming objects to the* n*-th target for* $n \in 1, \ldots, N$.

Using this lemma, we can compute the weights x_i that are the offsets of the robust hash routing function. Thus, the mapping algorithm now holds as:

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ x_j + h(\vec{v}, j) &= \max_{k \,\in\, [1,N]} x_k + h(\vec{v}, k). \end{aligned} \tag{6.5}$$

This solution does not require multiplication, only summation. However, designing a pseudo-random function with the probability distribution of hashing values equal to $\exp(-x)$ is not straightforward.

Lemma 2 presents the statement of the multiplier theorem only in another way. Clearly, if x_i and x'_i are weights computed with (2.3) and (2), S is the sum of the weights x'_i, and assuming that $\prod_{j=1}^{N} x_j = 1$, then

$$x'_i = \log x_i + S/N. \tag{6.6}$$

This property is vital because all the properties of the mapping and adaptation described in Chapter 4 are also valid for the set of weights x'_i. For example, thanks to Eq. (6.6), the adaptation of the weights x'_i will uphold the same minimal disruption property.

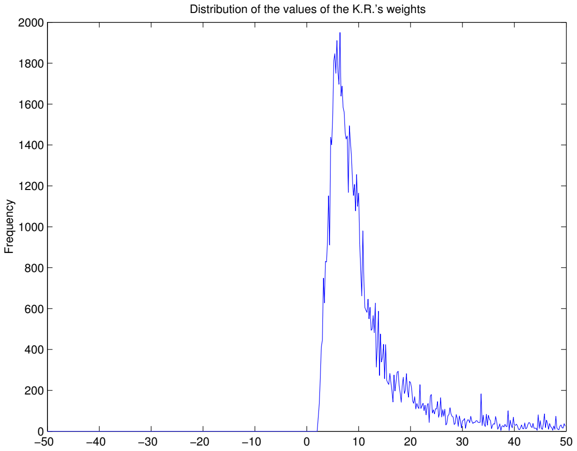


Figure 6.6: Ross' weights distribution.

Another important issue is understanding what range of values can such weights attain. The original weights x_i are always greater than zero. This does not hold for the transformed x'_i set. Because these weights must be loaded in the NP registers and we have to use a fixed-point notation to represent them, it is important to understand their possible distribution. In particular we need to know how many bits are needed for representing the integer part of the weight in order to guarantee that all possible percentage combinations (and thus the corresponding weights) are representable. We have conducted a C and Matlab simulation to anlayze the weights' values distribution. For $N\_CYCLES = 10000$, we randomly assigned percentages to the $N = 8$ ports and then computed the weights for both mapping algorithms. The results obtained are shown in Figs. 6.6 and 6.7.

The maximum and the minimum values of the weights appearing during the simulation (for (2.3) and (6.4) respectively) were

$$\text{Ross}: \left\{ \begin{array}{lcr} \max & = & 592.083951 \\ \min & = & 0.231048 \end{array} \right\} \qquad \text{Russo}: \left\{ \begin{array}{lcr} \max & = & 6.383648 \\ \min & = & -1.465130 \end{array} \right\}$$

We conclude that it is sufficient to use $1 + 4$ bits (instead of $0 + 10$) to represent the sign and the integer part of the weights described in (6.4).

As explained in Subsection 6.3.2, in order to avoid flow remappings altogether,

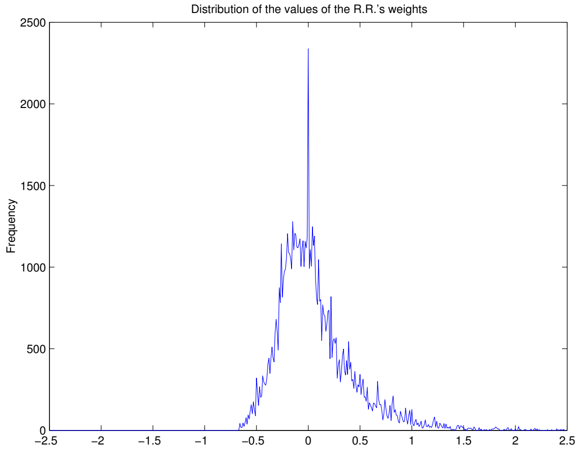


Figure 6.7: Russo's weights distribution.

the CP provides two sets of weights to the NP: the new weights x_k^{new} and the old weights x_k^{old}. The weight x_k may be positive or negative, and, contrary to the hash score which is always in the interval $[0, 1]$, we have to provide some bits for the integer part of x_k. Thus, it is non-trivial how to store the weights most efficiently.

Considering that the sum of weights (S) represents a degree of freedom, for any set of weights we can compute a set of weights which is equivalent and representable in our notation by only changing the value of S. Obviously, if a common offset is added to all the weights, the outcome of the HRW mapping will remain the same.

Thus, the weights can be stored, for example, in the 1-4-27 fixed-point notation, aligned for 32-bit processor arithmetic. This leads to a decimal precision of $2^{-27} \sim 7 \times 10^{-9}$, which should be precise enough for the possible weights adjustments. Sufficient precision is crucial for implementing a hierarchical weights storage data structure, as presented in Subsection 6.3.5.

As we need to add $score_i$ to the weight according to Eq. (6.5) and as we assume the weights are represented in a 1-4-27 (sign-integer-fractional) fixed point notation, we can use only the first 27 bits of the 32 produced by the uniform hash generator. Denoting by $a_0, a_1, \ldots, a_{30}, a_{31}$ the bits generated by the uniform generator, then the $score_i$ is computed as follows:

$$score_i = a_0 2^{-1} + a_1 2^{-2} + \ldots + a_{25} 2^{-26} + a_{26} 2^{-27}$$

6.3.5 HRW weights data structure

We use a table to store the weights on the NP. The NP can handle both static-size and dynamic-size tables. As we do not require that the table changes its size, and as accessing a dynamic-size table is slow (because it requires first to access a table that stores its size), it makes sense to use a static table to store the weights structure. Static tables on the NP do not have specific interfaces for setting and writing the data and thus we need to implement our own APIs. We can also choose the location in the memory for these tables: internal, SRAM or DRAM. The fastest memory is the internal, and we should use it because the table is read continuously. The internal NP RAM configuration, with 512 bits per line, guarantees that for each memory access, the NP reads 8 new and 8 corresponding old weights.

Flat data structure

Fig. 6.8 depicts the flow chart of the HRW mapping computation algorithm, as presented in Section 2.2.3. On two occasions, the execution waits for the results of the hash-coprocessor, which has been called before asynchronously earlier, to finish the desired computation. The algorithm contains two cycles: one for reading a 512-bit line from the memory and another one for computing the hash score for all the weights of that line. To implement these two cycles we used two indexes: M and N.

Using M and decrementing it each time by 8, the NP can parse one by one the N pairs of weights (x_k^{new}, x_k^{old}) contained in one line of the *Weight_Table*. M is used as offset in the load operand which reads the 32-bit block from the memory. The starting value of M depends on the number of weights expected to be on the line. For example, if $N = 12$ we need to parse two 512-bit lines: the first line contains 8 pairs of weights (and thus $M = 56$), while the remaining 4 pairs (and thus $M = (N-1) * 8 = 24$) are in the second line.

The initial value of N is the number of servers, but further on the variable is used for the hash score generation. The robust hash routing requires that an ID (or an index or a tag, etc.) of the server is inserted in the hash key, we can use N in the hash key, as it is decremented every time a pair of weights (x_k^{new}, x_k^{old}) is read. As N

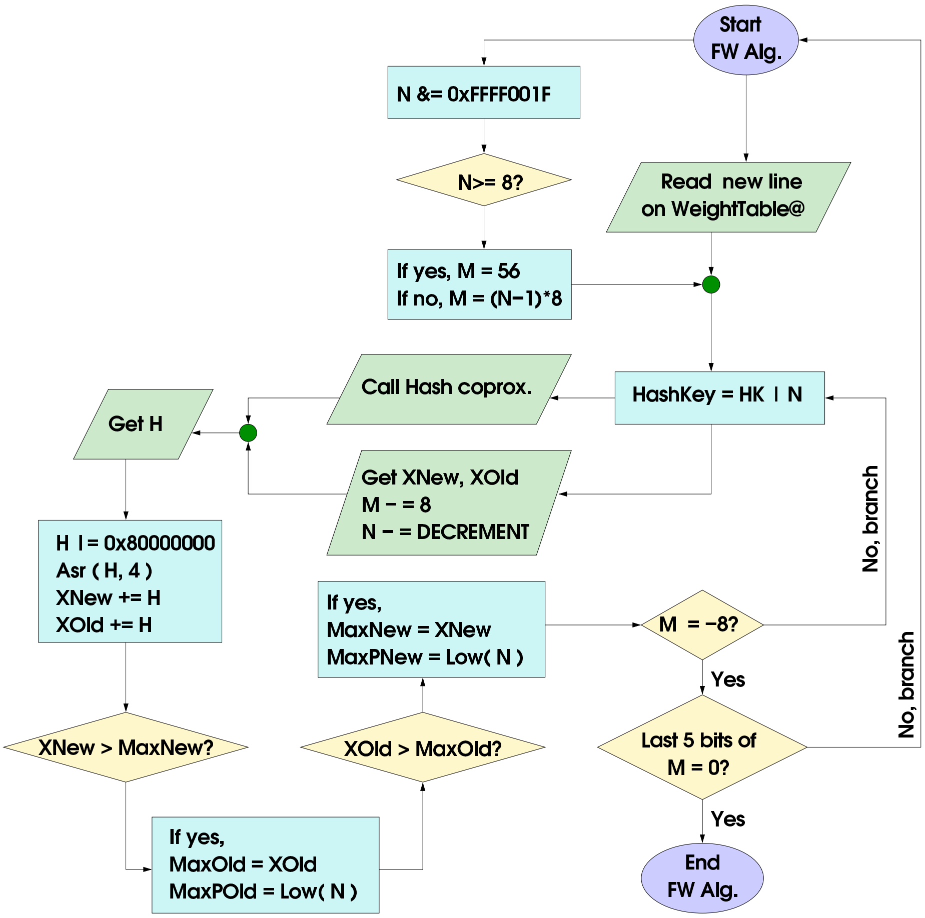


Figure 6.8: HRW Mapping computation algorithm

holds a different value per each server, it can be considered an ID.

Thus, for generating the hash scores it is sufficient to append N at the end of the hash key (the input to the hash coprocessor) as follows:

$$src\_IP@||src\_IP@ + src\_port << 8||N.$$

Unfortunately, the hash scores must be uncorrelated, and if only five bits are used for representing N, the hash coprocessor provides highly correlated results. Therefore, we use a 32-bit register for storing N and decrement it not by 1, but by a fixed 32-bit value, for example, the Fibonacci golden ratio (see Section 6.2.3). Thus, N

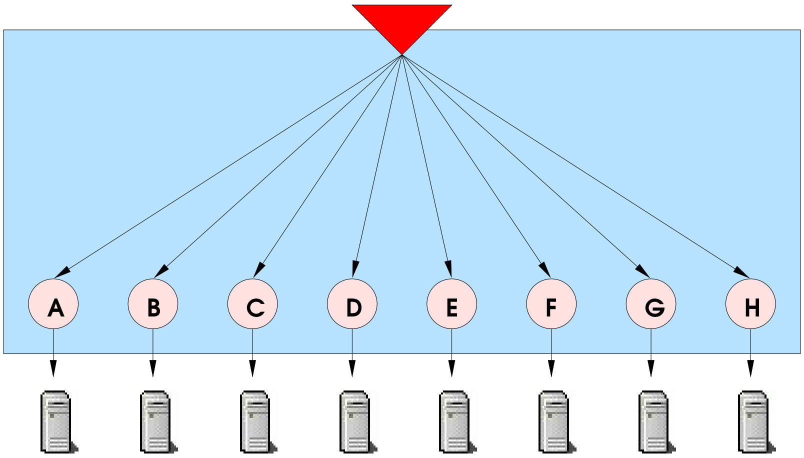


Figure 6.9: Flat data structure.

various indexes are generated, that, when plugged into the same hash key, result in independent hash scores.

Test results on the flat data structure

To test the algorithm implementation, we used a trace file [3] that logged the HTTP connections collected from 00:00:00 July 1, 1995 to 23:59:59 July 31, 1995, of the NASA Kennedy Space Center WWW server in Florida. There are about 260'000 different connections present in the log-file. This particular file has been chosen, even if being quite old, because the NASA server receives connections worldwide (we obtain a consistent view of the IP addresses spread) and because, for privacy reasons, it is now impossible to obtain trace files with real IP addresses (typically, IP addresses are encoded in the publicly available traces).

We simulated a server farm with three host (IDs: 120, 130, 140), having load percentages set to 20%, 30% and 50%. We sent 6496 UDP packets using the packet information (i.e. IP addresses) collected by the NASA server [3]. The results, summarized in Table 6.3, are close to desired, even if the number of packets sent is relatively small.

Table 6.3: Results of the simulation using a flat data structure.

ID	User percentage p_i (%)	Packets received	Percentage (%)
130	20	1160	17.86
140	30	2052	31.59
150	50	3284	50.55

Hierarchical data structure

Unfortunately, the complexity of the algorithm presented in Subsection 6.3.5 is linear with the number of servers N (we need to find the maximum among N different values). There are two cycles present: the inner one for parsing the weights inside a 512-bit line, computing the hash score and finding the maximum value, and the outer one for reading the 512-bit lines from the memory banks. Implementing these cycles means inserting branch operands that require many CPU cycles. Moreover, considering a server farm with, for example, 50 servers, it is difficult to set the 50 different percentages and it is hard to represent those because of the lack of precision.

One alternative to avoid the drawbacks above is to use a hierarchical approach, as illustrated by the following example:

$$\begin{aligned}\max(A,B,C,D,E,F,G) &= \max\left(\max(A,B,C,D),\ \max(E,F,G,H)\right)\\ &= \max\left(\max\left(\max(A,B),\ \max(C,D)\right),\ \max\left(\max(E,F),\ \max(G,H)\right)\right)\end{aligned}$$

This recursive approach changes the flat data structure (as shown in Fig. 6.9) into a multi-level hierarchical data structure (as shown in Fig. 6.10). The advantage is the lower computational complexity of the algorithm (logarithmic instead of linear). Unfortunately, the increased speed is traded for the increased space requirements, as well as for losing some of the algorithm's properties.

Firstly, *more than N weights* are needed now. There needs to be a weight present at every node of the decision tree. Let us illustrate that using an example (see Fig. 6.10): there are three levels of the decision tree, and the users must set the percentages of seven different *clusters*: those at the lowest level ((p_A, p_B), (p_C, p_D), (p_E, p_F) and (p_G, p_H)), those at the middle level ((p_{L11}, p_{L12}) and (p_{L21}, p_{L22})) and those at

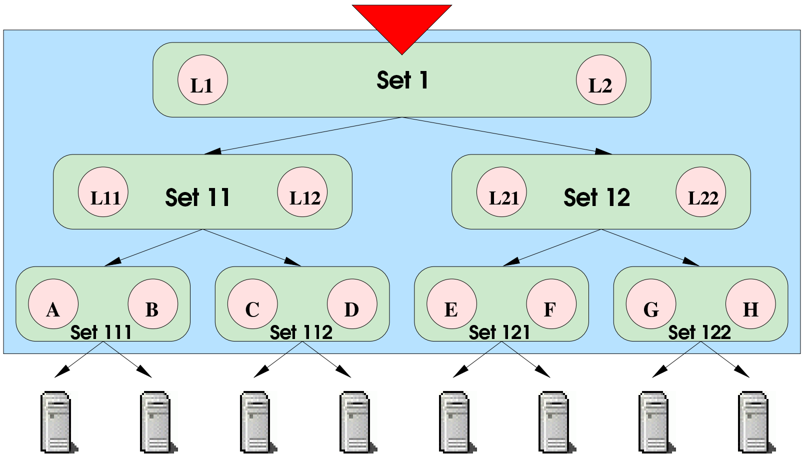


Figure 6.10: Hierarchical data structure

the top level $((p_{L1}, p_{L2}))$.

At each cluster, the sum of the fractions must be equal to 1. Server A now receives a fraction equal to $p_A \cdot p_{L11} \cdot p_{L1}$ of the incoming load. Such arrangement serves to provide better support for server heterogeneity–less powerful servers may now be combined into clusters, leading to more uniform load spreading.

Secondly, the CP must now execute *multiple different adaptation policies*, one per each cluster. For example, imagine that in cluster $Set11$ of Fig. 6.10, the mapping algorithm forwards too many packets to cluster $Set111$. What happens when the CP adapts the value of the weight x_{L11}? Some flows are redirected from cluster $Set111$ to cluster $Set112$, yet x_{L11} does not depend on x_A and x_B and thus some flows with real server A or B as original endpoint will be redirected to C or D, without changing x_A or x_B. Likewise, a change in x_{L1} (because too many flows arrive at $Set11$) will result in some flows originally directed to A, B, C and D to be redirected to E, F, G or H. Thus, changing at least one weight within a cluster will affect all the flows passing through that cluster.

In such a case, it is not possible to determine which particular flows are re-mapped. It can easily happen that some child clusters will have more flows re-mapped than others. Thus, adapting at level n could cause a cascade of adaptations for all clusters at lower levels .

The number of occurrences of this effect can be minimized by avoiding to carry out the adaptation at higher levels of the tree too often. This can be achieved by increasing the hysteresis bound in the adaptation policies related to the higher level clusters. When the higher levels are adapted, it is vital to avoid the adaptation on the lower levels for a certain period, to avoid cascades of adaptations.

The above considerations lead to the following set of rules when building the hierarchical structure:

- *Reasonable tree depth.* Too many levels aggravate the redirection problem, resulting in bad performance. Choosing typically 2, or at most 3 levels guarantees a fast execution and minimizes the redirection problem.

- *Reasonable number of branches.* It makes sense to group servers into autonomous sets of sufficient size. They must be neither too large, because of difficulties with setting the percentages and with weights representation, nor too small, because that increases depth of the tree unnecessarily. A reasonable maximal number of children of a node is 16, but a smaller number may be used as well. For example, when $N = 8$, the user can choose one level with 8 weights or 2 levels with 2 sets of 4 servers at the lower level.

Another implementation difficulty arises from the fact that in cases where the old and the new mappings differ (see Fig. 6.5), not only the packet, but also the new and the old result must be forwarded to the CP. Thus, knowing that TP^*_{New} is not equal to TP^*_{Old} is not sufficient, we must compute TP_{New} and TP_{Old} as well. The easiest way is to save all parameters concerning the old path and computing TP_{New}, then restoring them and computing TP_{Old}. Note that the algorithm thus must handle the special cases when TP_{New} and TP_{Old} are located at different tree depths.

Finally, it is vital to avoid correlation of hash keys between clusters. If the hash key for computing the scores inside two clusters (a father and a child) is always the same, it can happen that the result of the first cluster is highly correlated with that of the cluster-child, which is not desirable.

One option to avoid correlation is to add a 16-bit offset to the central part of the

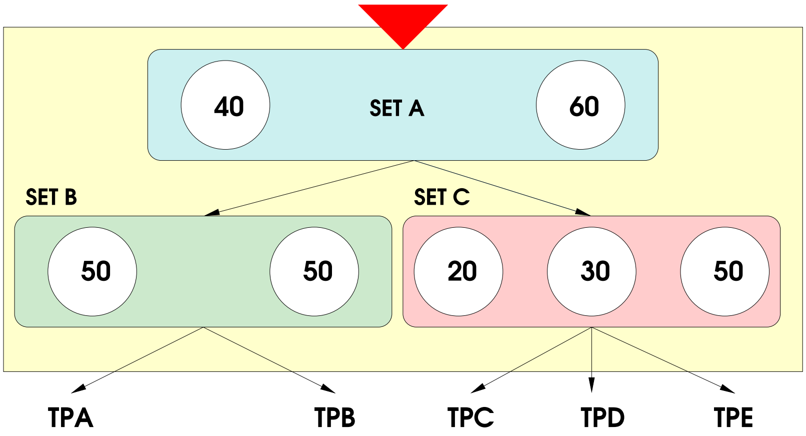


Figure 6.11: Topology for the hierarchical data structure simulations

hash key in order to un-correlate the results of different clusters. In our implementation we have used a constant `0xBEAF`, as there are many bits set to 1, leading to stronger un-correlation of the hash keys. Moreover, depending on the level, one may change the value of the decrement of N and thus further un-correlate the results.

Test results on the hierarchical data structure

We simulated a topology as shown in Fig. 6.11, and again the IP addresses collected by the NASA server [3] as input. We present the results of two simulations (see Table 6.4), and show how correlation between the results of different clusters can adversely affect performance of the method.

In the first simulation we used `0x9E377A|01` (the 24-bit Fibonacci golden ratio, see Subsection 6.2.3) as a decrement value for all clusters, and at the second level the first part of hash key was updated with $src\_IP@\|src\_IP@+src\_port << 8+0xBEAF <<$

Table 6.4: Results of two simulations on hierarchical data structure

ID	total % expected (p_i)	% obtained with Sim 1	% obtained with Sim 2
TPA	0.2	0.225	0.266
TPB	0.2	0.236	0.204
TPC	0.12	0.85	0
TPD	0.18	0.152	0.178
TPE	0.3	0.302	0.352

8. Even if the results are not as good as those obtained with the flat data structure, the difference between the desired and the obtained percentages is not overly significant.

In the second simulation, N was decremented by `0x9E377A|01` at the top level, while at the second level the decrement was `0x61c886|01` (the 24-bit complement of the Fibonacci golden ratio $1 - \phi^{-1}$), and the first part of the hash key was updated with $src\_IP@ \| src\_IP@ + src\_port << 8 + 0xBEAF << 8$. Clearly, this results in high correlation between the scores of clusters A and C. In fact, we observe that the server at TPC is never chosen. An optimal way of reducing correlation among clusters remains an open problem.

6.3.6 Network processor data plane implementation

Several software and hardware components of the PowerNP have been used in prototyping the load balancer application: standard layer-2, layer-3 and layer-4 (Multi-Field Classification) forwarding elements, as well as the hash function of the TSE coprocessor. The availability of these ready-made components made the data-path programming on the NP shrink to the relatively modest work of implementing the hash routing method, its managing API and the CP redirection. The speed and good spreading properties of the hash function in the TSE coprocessor enable us to consider the hash computation a black-box, eliminating the necessity to implement one's own hash function.

The number of processor instructions required to execute the hash routing method on each packet is dependent on the number of balanced servers M. The instructions are primarily dedicated to reading and carrying out operations on the per-server weights, while calling the TSE coprocessor in parallel. Up to $M = 8$ the prototype implementation requires executing $75 + M * 20$ instructions. For $M > 8$, the number of instructions executed grows logarithmically with M, as the weights' table is then organized into a tree structure.

6.4 Conclusions

In this chapter, we have demonstrated the practical applicability of the adaptive load-sharing method.

First, we have presented two examples of router architectures that would significantly benefit from executing the load-sharing method among the processors present within a router. We have discussed some key implementation issues, such as the pseudo-random function computation and the load information gathering and proposed suitable alternatives.

In the second part of the chapter, we have described and documented an existing implementation of the load-sharing method on the IBM PowerNP network processor. In this scenario, the network processor acts as a server farm load balancer. We have addressed some specific problems with implementing the HRW mapping computation and data structures. The parallelism of the network processor is exploited in order to provide a fast solution and to extend the method in order to avoid flow remappings altogether.

Chapter 7

Conclusions

7.1 Open issues

Further insight needs to be acquired into the impact of remote packet information processing—the influence of the information vector round-trip time on packet delay within a multiprocessor system. A related aspect to study is the ability to assign a preference to locally attached processors over remotely located ones.

Another open issue is the dissemination and gathering of load information. In the scenarios presented in this thesis, feedback information gathering has been performed by a central entity–the router control point or the server farm load balancer–which are natural points for maintaining centralized control over the entire multiprocessor system and would have been in an equivalent position without the load-sharing method in operation. However, one may well imagine a fully distributed multiprocessor environment without a controlling entity. Devising a specific distributed version of the adaptation algorithm for such a purpose remains a challenge.

When a load-sharing method of the kind described here is deployed, it may in fact be exploited to achieve even further performance benefits than those described in this thesis. Given the fact that the knowledge about what traffic is mapped where can easily be made available not only to the entity that performs the mapping, but also to other elements within the system, in particular the processing units, one can envision a system where the processing units adjust their processing methods to the locality enforced on the incoming traffic by the mapping. Significant further performance gains may be achieved by such partitioning of the problem space.

7.2 Concluding remarks

In this work, we have demonstrated the advantages of deploying a load-sharing method in a multiprocessor system that acts as a node in a networked environment. We have presented an adaptive load-sharing method and demonstrated its favorable properties, both by theoretical means and by simulations. In addition, we have presented two practical examples, a router architecture and server load balancer implementation, of the method's application.

The value of the proved minimal disruption property of the mapping adaptation has been demonstrated in the extensive set of simulations. At the negligible cost of remapping on average less than 0.1% of all flows, the method reduces the probability of a packet loss on average by 60% in comparison to a static, non-adaptive load-sharing method. Furthermore, the remappings can be avoided altogether by a simple additional classification mechanism.

Such a scheme is particularly useful in systems with many input ports and packets requiring large amounts of processing. With the proposed scheme, a kind of statistical multiplexing of the incoming traffic over the multiple processors is achieved, thus in effect transforming a network node into a parallel computer. The improvements of processor utilization decrease the total system cost and power consumption, as well as improve fault tolerance.

The spectrum of potential applications is not limited to a router or a server farm. The method can be deployed in any networking system that benefits from spreading the load over multiple processing units and that requires packets belonging to a single flow to be processed by the same processor. Such applications include, for example, a distributor of traffic over multiple network links or a distributed caching mechanism.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] AIX–MAE West interconnection at NASA Ames OC-3 trace. National Laboratory for Applied Network Research (NLANR), March 19 2000. http://moat.nlanr.net/PMA/.

[3] The Internet Traffic Archive. Trace file of NASA Kennedy Space Center WWW server in Florida, 2002. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.

[4] A. Asthana, C. Delph, H. V. Jagadish, and P. Krzyzanowski. Towards a Gigabit IP router. *Journal of High Speed Networks*, 1(4):281–288, 1992.

[5] G. Barish and K. Obraczka. World Wide Web caching: trends and techniques. *IEEE Communications Magazine*, 38(5):178–184, May 2000.

[6] E. Basturk, R. Engel, R. Haas, V. Peris, and D. Saha. Using network layer anycast for load distribution in the Internet. In *Proceedings of the Global Internet Symposium*, Sydney, 1998.

[7] Z. Cao, Z. Wang, and E. W. Zegura. Performance of hashing-based schemes for Internet load balancing. In *Proceedings of the INFOCOM'00 Conference*, pages 332–341, 2000.

[8] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[9] Cisco Express Forwarding (CEF). Cisco Systems white paper, 1997.
http://www.cisco.com.

[10] H. C. B. Chan, H. M. Alnuweiri, and V. C. M. Leung. A framework for optimizing the cost and performance of next-generation IP routers. *IEEE Journal on Selected Areas in Communications*, 17(6):1013–1029, June 1999.

[11] H. J. Chao. Next generation routers. *Proceedings of the IEEE*, 90(9):1518–1558, September 2002.

[12] Internet Software Consortium. Berkeley Internet Name Domain (BIND), May 2002.
http://www.isc.org/products/BIND/.

[13] M. Crovella, M. Taqqu, and A. Bestavros. *A Practical Guide To Heavy Tails*, chapter 1: Heavy-Tailed Probability Distributions in the World Wide Web, pages 3–26. Chapman & Hall, 1998.

[14] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *ACM Computer Communication Review*, 27(4):3–14, October 1997.

[15] P. A. Dinda and D. R. O'Hallaron. An evaluation of linear models for host load prediction. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, CA, August 1999.

[16] G. Dittmann and A. Herkersdorf. Network processor load balancing for high-speed links. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2002)*, pages 727–735, San Diego, CA, July 2002.

[17] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogenous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[18] F. Baker (editor). Requirements for IP version 4 routers. RFC 1812, Internet Engineering Task Force, June 1995.

[19] H. El-Rewini, H. H. Ali, and T. Lewis. Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37, December 1995.

[20] A. Engbersen and C. Minkenberg. A combined input and output queued packet-switched system based on a Prizma switch-on-a-chip technology. *IEEE Communications Magazine*, 38(12):70–77, December 2000.

[21] C. Partridge et al. A fifty gigabit per second IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, June 1998.

[22] f5 Networks. BIG-IP intelligent load balancing, 2002.
http://www.f5.com/f5products/bigip/LB520/.

[23] G. C. Fedorkow. Cisco 10000 Edge Services Router (ESR) technology overview, 2000.
http://www.cisco.com.

[24] G. Goldszmidt and G. Hunt. Scaling internet services by dynamic allocation of connections. In *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, pages 171–184, May 1999.

[25] I. Iliadis and W. Denzel. Performance of a packet switch with input and output queueing under unbalanced traffic. In *Proceedings of the IEEE INFOCOM 1992*, pages 743–752, May 1992.

[26] I. Iliadis and W. Denzel. Analysis of packet switches with input and output queueing. *IEEE Trans. on Communications*, 41(5):731–740, May 1993.

[27] Juniper Networks Inc. M160 Internet backbone router datasheet, August 2001. http://www.juniper.net.

[28] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[29] R. Jenkins. Details of some hash functions and block ciphers, 2002. http://www.burtleburtle.net/bob/hash/index.html.

[30] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Trans. on Communications*, 35(12):1347–1356, December 1987.

[31] L. Kleinrock. *Queuing Systems*, volume 3. John Whiley & Sons, 1975.

[32] D. E. Knuth. *Sorting and searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, second edition, 1998.

[33] O. G. Koufopavlou, A. N. Tantawy, and M. Zitterbart. Analysis of TCP/IP for high performance parallel implementations. In *17th IEEE Conference on Local Computer Networks*, Minneapolis, September 1992.

[34] V. P. Kumar, T. V. Lakshman, and D. Stilliadis. Beyond best effort: router architectures for the differentiated services of tommorrow's Internet. *IEEE Communications Magazine*, pages 152–164, May 1998.

[35] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[36] T. Kurz, P. Thiran, and J-Y. Le Boudec. Regulation of a connection admission control algorithm. In *Proceedings of INFOCOM'99*, pages 1053–1061, New York, March 1999.

[37] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[38] K. McCloghrie and M.T. Rose. Management information base for network management of TCP/IP-based internets:MIB-II. RFC 1213, Internet Engineering Task Force, March 1991.

[39] N. McKeown, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switches. In *Proceedings of the 1996 IEEE INFOCOM*, pages 296–302, San Francisco, CA, March 1996.

[40] N. McKeown and B. Prabhakar. High performance switches and routers: Theory and practice. In *Tutorial M2 of the 1999 ACM SIGCOMM*, Cambridge, MA, August 1999.

[41] Microsoft. Windows NT Load Balancing Service, July 2001.
http://www.microsoft.com/ntserver/ProductInfo/features/WlbsFeat.asp.

[42] Nortel Networks. Alteon portfolio.
http://www.nortelnetworks.com/products/01/alteon/, 2002.

[43] P. Newman, G. Minshall, T. Lyon, and L. Huston. IP switching and Gigabit routers. *IEEE Communications Magazine*, January 1997.

[44] S. Nilsson and G. Karlsson. Fast address lookup for Internet router. In *Proceedings IFIP 4th International Conference on Broadband Communications '98*, pages 11–22, 1998.

[45] WAN traffic distribution by address size, Fix-West trace. National Laboratory for Applied Network Research (NLANR), May 1997.
http://www.nlanr.net/NA/Learn/Class.

[46] Z. Opalka and S. Soman. Will the new super routers have what it takes? Nexabit Networks White Paper, March 1999.

[47] R. Perlman. *Interconnections - bridges and routers.* Addison-Wesley, 1992.

[48] Resonate. Central Dispatch, 2002.
http://www.resonate.com/solutions/products/.

[49] Jennifer Rexford, John Hall, and Kang G. Shin. A router architecture for real-time communication in multicomputer networks. *IEEE Transactions on Computers*, 47(10):1088–1101, October 1998.

[50] K. W. Ross. Hash routing for collections of shared web caches. *IEEE Network*, 11(6):37–44, November-December 1997.

[51] Ch. Semeria. Internet backbone routers and evolving Internet design. Juniper Networks white paper, September 1999.
http://www.juniper.net.

[52] J. Sethuraman and M. S. Squillante. Optimal stochastic scheduling in multiclass parallel queues. Performance Evaluation Review Vol. 27 1, MIT, 1999.

[53] A. Shaikh, J. Rexford, and K. G. Shin. Load-sensitive routing of long-lived IP flows. In *Proceedings of SIGCOMM '99*, September 1999.

[54] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the effectiveness of DNS-based server selection. In *Proceedings of the IEEE INFOCOM 2001*, Anchorage, AK 2001.

[55] B. A. Shirazi, A. R. Hurson, and K. M. Kavi (editors). *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE CS Press, 1995.

[56] Akamai Web site.
http://www.akamai.com/.

[57] Speedera Web site.
http://www.speedera.com/.

[58] Cisco Systems. Cisco content networking devices, 2002.
http://www.cisco.com/en/US/products/hw/contnetw/index.html.

[59] Cisco Systems. Overview: Load balancing with the multinode load balancing (MNLB) feature set for LocalDirector, 2002.
http://www.cisco.com.

[60] Coyote Point Systems. Equalizer, 2002.
http://www.coyotepoint.com/equalizer.htm.

[61] A. Tantawy and M. Zitterbart. Multiprocessing in high performance IP routers. In *Proceedings of the 3rd IFIP WG 6.1/6.4 Workshop on Protocols for High Speed Networks*, Stockholm, May 1992.

[62] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, February 1998.

[63] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–27, November-December 1997.

[64] J. Turner and T. Wolf. Design issues for high performance active routers. *IEEE JSAC*, 19(3):404–409, March 2001.

[65] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP route lookups. In *Proceedings of ACM Sigcomm '97*, pages 25–37, Cannes, France, 1997.

[66] T. Wang. Integer hash functions: principles and solutions, August 2002.
http://www.concentric.net/∼Ttwang/tech/inthash.htm.

[67] X. Xiao and L. Ni. Parallel routing table computation for scalable IP routers. In *Proceedings of the IEEE International Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, pages 144–158, Las Vegas, February 1998. Published by Springer as Lecture Notes in Computer Science 1362.

[68] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. In *Proceedings of ACM Sigcomm '02*, Pittsburgh, August 2002.

[69] H. Zhu, T. Yang, Q. Zheng, D. Watson, O. H. Ibarra, and T. Smith. Adaptive load sharing for clustered digital library servers. In *Proceedings of the Seventh*

International Symposium on High Performance Distributed Computing, pages 235–242, July 1998.

Appendix A

List of publications and Curriculum Vitae

Journal papers

- L. Kencl, J. Y. Le Boudec. "Adaptive Load Sharing for Network Processors". *IEEE/ACM Transactions on Networking*, in submission.
- R. Haas, P. Droz, L. Kencl, A. Kind, B. Metzler, C. Jeffries, R. Pletka, M. Waldvogel. "Creating Advanced Functions on Network Processors: Experience and Perspectives". *IEEE Network*, in submission.

Conference papers

- L. Kencl, J. Y. Le Boudec. "Adaptive Load Sharing for Network Processors". In *Proceedings of the IEEE Infocom 2002*, New York, June 2002.
- E. Bowen, C. Jeffries, L. Kencl, A. Kind, R. Pletka. "Bandwidth Allocation for Non-Responsive Flows with Active Queue Management". *International Zurich Seminar on Broadband Communications 2002*, Zurich, Switzerland, February 2002.

- L. Kencl, B. Radunovic. "General Method for Finding the Most Economical Distributed Router Architecture". In *Proceedings of the Communication Networks and Distributed Systems Modelling and Simulation Conference 2002*, San Antonio, Texas, January 2002.

Research reports

- R. Russo, L. Kencl, B. Metzler, P. Droz. "Scalable and Adaptive Load Balancing on IBM PowerNP". *IBM Research Report No. RZ-3431*, July 2002.

Patents

- L. Kencl, E. Bowen, P. Droz, B. Metzler. "Load Balancing in Data Networks". *European Patent Office Application No: 02014116.4*, June 2002.
- P. Droz, L. Kencl. "Method and System for Processing Data Packets". *European Patent Office Application No: 00114246.2*, July 2000.

Curriculum Vitae

Personal

Name: Lukas KENCL
Date of birth: January 2, 1970
Place of birth: Prague
Nationality: Czech Republic
Marital status: married

Contact

Work Address: IBM Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon
Switzerland
Tel.: (+41)-1-724 8426
E-mail: lke@zurich.ibm.com
lukas.kencl@ieee.org

Private Address: Isengrundstrasse 12
CH-8134 Adliswil
Switzerland

Employment history

1999 - present	*Researcher* Network Processor Software Group Department of Communication Systems IBM Zurich Research Laboratory
1998	*Network Design Specialist* Technical Division Eurotel s.r.o., Prague
1997	*Oracle Express Consultant* VUMS Software a.s., Prague
1997	*Assistant* Department of Mathematics Prague Institute of Chemical Technology

Education

1999 - present	*Ph.D. Candidate* Laboratory for computer Communication and Applications (LCA) Swiss Federal Institute of Technology (EPFL) Lausanne, Switzerland
2000 - 01	*Micro MBA Program* MBA courses in Finance, Marketing and Strategy successful finish in June 2001.
1997 - 98	*Graduate School of Communication Systems* EPFL, Lausanne successful finish in July 1998 Project: IP Routing Address Lookup Algorithms Evaluation
1988 - 95	*School of Computer Science* Faculty of Mathematics and Physics Charles University, Prague graduated M.S. in Computer Science in June 1995 Thesis: Approximations of a Maximum Cut of a Graph