RZ 3431 (# 93699) 07/08/02
Computer Science 106 pages

Research Report

Scalable and Adaptive Load Balancing on IBM Power NP

Riccardo Russo*, Lukas Kencl, Bernard Metzler, and Patrick Droz

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

*Permanent address: Institut Eurécom, 06904 Sophia-Antipolis, France

Scalable and Adaptive Load Balancing on IBM Power NP

Riccardo Russo*, Lukas Kencl, Bernard Metzler, and Patrick Droz

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

*Permanent address: *Institut Eurécom, 06904 Sophia-Antipolis, France*

Abstract

Web and other Internet-based server farms are a critical company resource. A solution to the increased complexity of server farms and to the need to improve the server performance in terms of scalability, fault tolerance and management is to implement a load balancing technique. It consists of a front-end machine which intelligently redirects the traffic to several Real Servers. We discuss the feasibility of implementing adaptive load balancing with minimal flow disruption on the IBM PowerNP Network Processor. We focus our attention on the steady-state part of the algorithm and propose a PowerNP-tailored mapping algorithm derived by Robust Hash Mapping. We propose and show a fast algorithm solution (despite the simple arithmetical logic of the PowerNP), as well as a scalable approach (aiming at minimizing the packet processing time) and, finally, we present some initial performance results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Connection speed, latency, fault tolerance, and ease of management are becoming the keywords for the new web server architectures. Because the constant upgrading of server capacity is not an economically-sound solution, several techniques have been implemented in the past few-years in order to provide a better service to the internet users.

This thesis concerns the implementation of a load balancing algorithm on the IBM PowerNP which answers the needs of the market completely. The algorithm is composed of two parts: packet forwarding and link adaptation. While the former one spreads the connections to several real servers based on administrator-defined link percentages, the latter guarantees high link utilization. We will focus mainly on the packet-forwarding part. We will propose a feasible algorithm, based on Robust Hash Mapping, and we will show its implementation, and provide real simulations showing the correctness of our choices.

In this chapter, we will understand what the architectural requirements fitting the market needs are, and we will show what the current solutions on the market are. In particular, we will focus our attention on those that implement load balancing because they are the preferred choice of network administrators for their easiness and efficacy. Our solution will also be briefly presented.

1.1 Architectural demands

In recent years, the phenomenal growth of the Internet/Extranet/Intranet (Xnet) and the need to conduct E-commerce and business over the World Wide Web (WWW) created new challenges for network administrators. As a web site filed more and more clients, the first traditional solution, i.e. the increasing of capacity of the web servers, was neither economically sound nor scalable. Even if we install more RAM on existing machines, replace the CPUs with a faster one, use faster or dedicated SCSI controllers and disks with shorter access times (RAID systems), as the server's maximum capacity is reached, response times increase until users are refused access or will not accept the degradation of performance. Thus, the web sites begin to lose clients and money. In fact, as stated by Zona Research [1] in 1999:

> *The 32% of the Internet users having problems in accessing a Web Site, stop to use it and look for an alternative and they unlikely will try to open it another time.*

Nowadays, the situation is still the same. Forrester Research [2] said in 2001:

> *The 58% of Web Site users said download speed was a primary factor that influences them to return to a site.*

It is certainly evident that the upgrading techniques described above are as costly as they are inefficient. This kind of solution is not scalable at all: the administrators are forced to buy newer and increasingly large-scale machines to satisfy the users' needs. This means large immediate expenses for buying a product that is always at the forefront of technology. In contrast the market requires scalable solutions that are products (hardware or software) which continue to function well as they (or their context) are changed in size or volume. But the requests of the market do not end there. Administrators require, for example, flexible products to avoid becoming trapped in a fixed network configuration. The market also demands products which are easy to manage and can be easily integrated into existent systems. 24-7 fault-tolerant systems are specially requested because the web servers must be

able to operate not only in any traffic condition (including hostile traffic such as hacker attacks) but also if there are unexpected internal problems (i.e. software crashes or CPU melt down). This is the opinion of John Dodds, Senior Administrator of Financialweb.com interviewed by InformationWeek [2] about the "no down time" concept: *"Now, it's absolutely critical to be up 100% of the time. We have some real-time traders as customers, and for some of them, just five-second delay can amount to 15% profit or 15% loss".* These requirements are vital for offering high-quality services to a huge number of possible clients. A lot of money has been invested, but also the worldwide losses [2] caused by heavy traffic problems are huge:

- $58 millions a month are lost owing to web page loading failures
- $3 millions a month are at risk in securities industry owing to unacceptable download times
- $2.8 millions a month are at risk in travel industry owing to slow download times
- ... and the list goes on for a total of $4 billion per year.

It is clear that new solutions are needed to solve these new challenges, and in the next paragraphs we will present them.

1.2 Recent solutions

As constant upgrading does not assure any of the critical missions of the new web sites, other solutions have been proposed. Load Balancing, Web Caching and the Content Delivery Networks are the ones most requested. Before analyzing the load balancers already launched on the market in detail, we will provide a general overview of the other two solutions.

1.2.1 Caching and Content Delivery Networks

With **Web Caching**, we use 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-redirectors. If a cache does not have 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 onto itself on its way back to the client. This means that the proxy server can provide static content from its cache itself, when the request is repeated.

Content Delivery Networks (CDNs) are private networks of geographically dispersed caching servers at the edge of the Internet. They bring content (i.e. multimedia streaming) closer to users and speed up content delivery. The main providers are Akamai [3], Speedera [4], and Digital Island [5]. Akamai, for example, has nearly 12,000 servers in 62 countries. In Europe there is the Overnet network managed by Madge.Web [6]. The CDNs are thus 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 originating 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.

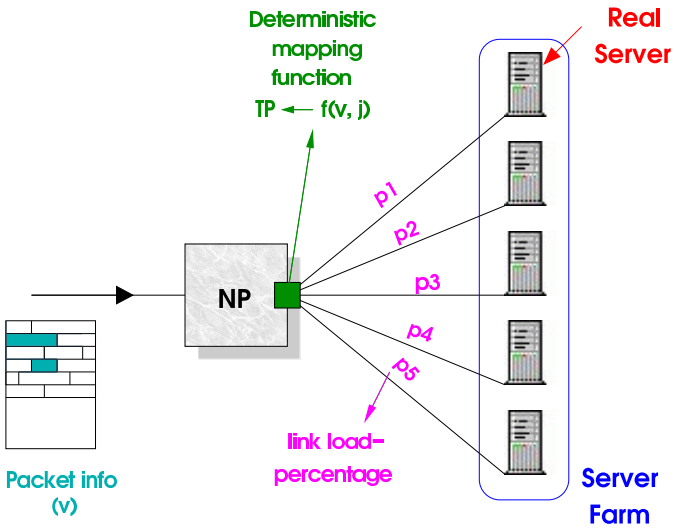

Figure 1.1: Topology for load balancing

1.2.2 Load balancing

This is the most common technique. As shown in Fig. 1.1 a pool (called Server Farm) of identical Web Servers is connected with the Internet by a load balancer that acts as a front-end machine which intelligently directs the clients (their TCP connection) to the Real Servers according to the servers' capacities and status. This allows each server to operate more efficiently. The choice of the Real Server falls into two basic categories: *content unaware* and *content aware*.

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

In the second case the load balancer is an end-point of the TCP connection, and can understand the user's request through a parsing of layer-7 information (i.e URL). The redirection happens after the three-way handshake between the load balancer and the user. Now all real servers can for example have different contents or different levels of security or power.

This solution is more flexible but less fast than the previous one considering that the load balancer must be able to parse layer-7 information spread over several packets. Another advantage of this solution is that is able to manage *sticky connections*. Some applications require that some user connections are always redirected to the same real server. Examples are filling of forms, shopping carts, bank transactions, payments, etc. These particular connections are typically handled in three ways:

- The load balancer saves into a table the couple user and IP address. This solution does not work with the Network Address Translation (NAT) because for each new connection of the same user, there is a different, and dynamically assigned, source IP address.
- The real server or the load balancer understand that is a sticky-connection and sends a cookie back to the user that will be used next time to know where to redirect the connection. It does not work with encrypted connections.
- The real server or the load balancer uses HTTP to redirect to the correct real server.

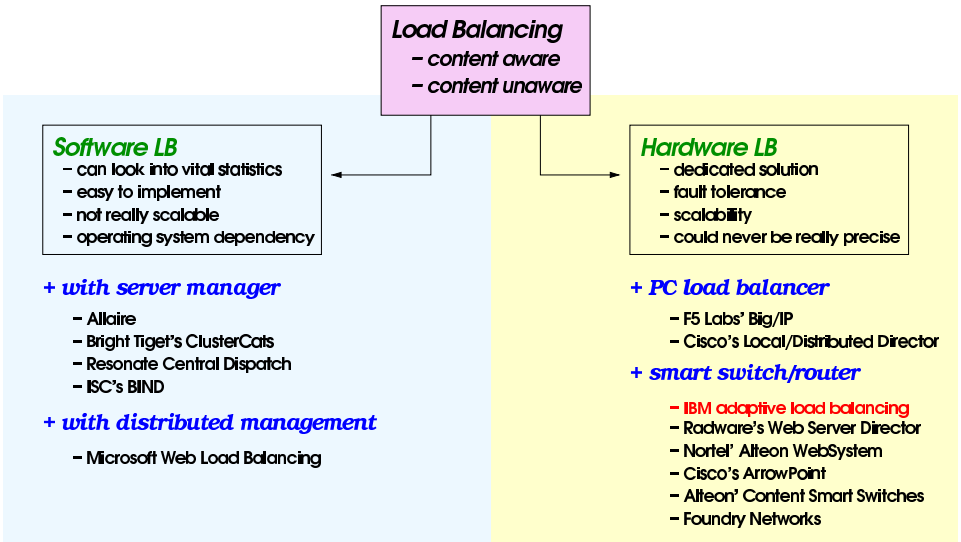

Figure 1.2: Load-balancing categories and products

In the last part of this section, we will discuss some of the most interest-

ing load-balancing solutions already on the market. As shown in Fig. 1.2, all of them fall into one of two basic large categories: software and hardware. We will analyze these categories, and provide a general description of the main products.

Software solutions

To save money, many network administrators use a readily available software load balancer which resides in the primary web server and intercepts packets as they enter the site. The first solution is the DNS Round Robin. The state of the art in DNS-server implementation is the Berkeley Internet Name Daemon (*BIND*) developed by Internet Software Consortium (ISC). It let us to select and provide a particular IP address from a pool of addresses when a DNS request arrives. The selection pointer of this pool is incremeneted and when it reaches the last element, it starts again with the first one. The DNS server is configured using multiple CNAME (canonical name) resource records. Unfortunately this solution has numerous drawbacks:

- It does not know whether the skipped server is down or busy and does not remove failed servers from the rotation.
- Even if the load balancer spreads the connections equally, this has no relationship to the load of connections. One server could be really busy, and others not.
- If the primary web server goes down we have a problem.
- 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 different size/power.
- With big server farms, DNS Round Robin is difficult to configure.

The BIND solution is really simple and it is quite old. More recent load balancers are more complex. Looking at the products on the market,

we can see that they fall in two categories: they can have a *server manager* like Allaire, Bright Tiger (ClusterCats) and Resonate Central Dispatch, or a *distributed management* as Microsoft Web Load Balancing Service (WLBS). Let us examine one solution from each category.

The Resonate Central Dispatch, as described in Ref. [7], is a software-based server management solution. It proposes 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. This ensures that end-user connections go through even if a process hangs, improving server availability.

WLBS, as described in Ref. [8], distributes the incoming load of IP requests across multiple nodes running the Network Load Balancing software. The load-balancing scheme is based on a fast hashing algorithm 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. This technology comes for free with Windows NT.

Even if they seem to be really powerful solutions, all these software-based solution have some common drawbacks:

overhead : even if they can look into vital statistics on the server, such as CPU or memory utilization, the extra software might use up an unknown amount of resources on that server. If the task is rich then a great percentage of server power is depleted for the load-balancing tasks.

stability : failure of the load-balancing software could cause a system crash.

scalability : load-balancing software has to be installed for every new server added to the cluster. Moreover, there are possible "per server" licensing costs.

system dependency : the software can restrict a network to a single operating system without any chance for heterogeneity between the servers.

Hardware solutions

In this case companies propose a stand-alone unit (i.e. a "black box") that is physically positioned between the user community (Internet) and the server cluster and handles the entire flow between them. The advantage is that there is a dedicated hardware for a dedicated task, which typically entails operating system independence, allowing the platform to operate in a heterogeneous server environment. This makes the typical hardware solution very flexible. Moreover the overall maintenance of the hardware load balancers is simple and can be done with minimum effort. It can easily be replaced with a newer/better solution. For upgrades and periodic maintenance only a single device in the network has to be tampered with, not every server. However there is a single point of failure, and this is a potential hazard but it is only a concern if dual and redundant units, configured to activate in case the main unit fails, are not used. However a strictly hardware solution could never be really precise. In fact it is infeasible to gather information such as CPU/memory utilization without software agents.

The hardware solutions are grouped into two big categories: *PC load balancer* (also called *Appliance Load Balancer*) and *smart switch/router*.

The **PC load balancer category** is a software solution put into a PC and sold as a package. This offers some router-like functionalities and saves the time and effort of integrating a software solution with the existing equipment. Also, there are many failure points in a PC, far more than found in a switch or router. Examples of commercially available PC-based load balancing are F5 Labs' BIG/ip and Cisco's Local Director and Distributed

Director. Let us describe these two products.

The F5 Labs' BIG/ip, as described in Ref. [9], is situated between the network and server farm, and continuously monitors each server for service availability and performance. It automatically routes incoming queries to the most available server. It also provides support for heterogeneous server farms. With BIG/ip, the network administrators can also balance web, e-mail, database, FTP and/or firewall traffic to ensure content availability for end users. BIG/ip accepts all data packets addressed to the site's IP address and distributes them to the most available server. The supported load-balancing algorithms are round robin, ratio, least connections, fastest, priority, predictive, and observed. It is a completely agent-free technology, no extra software is needed.

With Cisco's Local Director, as described in Ref. [10], we can set up a virtual server that can bounce requests to one or more Web servers behind the Local Director. Probes (agents), which monitor the health of servers in the background, can sound an alarm for a failed server and redirect requests to the remaining Web servers. Specific agents, created by third-party developers, can also be integrated into the system. Moreover the servers can be automatically and transparently placed in or out of service, and the Local Director itself is equipped with a hot standby fail-over mechanism, eliminating all points of failure for the server farm. The Cisco Local Director is tightly integrated with the Cisco network. For example, it can work in conjunction with the Catalyst® 6000 and Catalyst 6500 to accelerate TCP sessions. The load-balancing decisions are based on several algorithm: application availability, server capacity, round robin, least connections, or Dynamic Feedback Protocol (DFP).

The **smart switch/router category** provides the best integration with the existing network; in fact the switch/router works with any O/S or platform, is robust and rarely fails. Common examples are Web Server Director (WSD) by RADWARE, Alteon WebSystem by Nortel, ArrowPoint by Cisco, Content Smart Switches by Alteon Networks and Foundry Networks. All

these products are quite similar. Let us describe only RADWARE, which is the leader of this segment.

As described in Ref. [11], in addition to traffic load balancing, all the sites protected by the WSD devices can also have firewall-like security features, full monitoring, statistics, and full fault tolerance. The collection of load-balancing policies is really large and, in particular, it includes the designated back-up server algorithm. They created two basic versions. In the Network Proximity version, they combine proximity-based geographic redirection and intelligent load balancing on a single platform, whereas in the Distributed Sites version, they guarantee a reliable and efficient connection for local/remote servers and remote sites, even if they are down or busy.

1.2.3 IBM PowerNP based load balancing

The IBM Web Server load balancing will be a feature of the PowerNP Network Processor, which is a programmable router with special units (the Control Point and Coprocessors) running all functions requiring special processing and logic. Thus, this solution inherits all the qualities described for the smart router category (fault tolerance, robustness, scalability, security) but maintains also the capacity of performing complex operations.

The administrator only has to set the load percentages of each real server, and a forwarding algorithm will send the requested amount of load to each Real Server. Based on the Robust Hash Mapping idea, we created a fast hash function, currently based only on layer-3/4 information. Thanks to its properties, real servers can easily be added or removed from the server farm. Moreover, a control loop monitors all the link utilizations and guarantees high performance for all real servers. Scalability is provided by the multi-level configuration, which allows the administrator to create cascades of server farm. High speed is provided by Network Processor parallelism. Stability and ease of management are also guaranteed by the IBM technology and support.

1.3 Organization of the thesis

In this thesis we will focus our attention on the creation and implementation of the forwarding part of the proposed load balancing algorithm on the PowerNP picocode. For this reason, in the second chapter we will provide a general overview of the Network Processor architecture, introducing all devices and internal mechanisms that are essential for understanding our choices and the final realization. In the third chapter, which is the longest, we will discuss the forwarding function in detail. We will explain the problems of its implementation on the PowerNP, and after having modified the function, we will obtain a version which can be implemented on PowerNP. Matlab and java simulations, tests, and graphical approaches will help us to evaluate every single block and the forwarding-algorithm behavior together with the adaptation policy. The fourth chapter provides details on the real implementation, network topology, and real simulations, and, finally presents a scalable approach which clearly increases the speed of the entire algorithm. The thesis closes with final the conclusions on the last chapter.

Chapter 2

IBM Network Processor overview

2.1 The IBM Network Processor

Network Processors (NPs) are expected to become the fundamental network building block for networks in the same fashion that microprocessors are for today's PCs. NPs are dedicated processors designed for flexible packet processing at wire speed.

The IBM Network Processor [12] is a programmable switching and routing system on a single chip which is meant to scale from small to large applications and to serve a huge number of users. It provides wire-speed frame processing and high forwarding and filtering capability thanks to many functions that are incorporated into the system such as Hardware Accelerators, Coprocessors and Parallel-Thread Processing. As a programmable integrated communications circuit or, better, a 'programmable ASIC', the Network Processor provides very efficient packet classification, multiple table-lookups per frame, packet modification, queue/policy management, and other packet-processing capabilities. Moreover, the programmable features of the NPs guarantee a flexible device to the network product developers for leveraging their investments because they can implement new protocols and technologies without having to create new custom ASIC designs or to perform hardware modifications.

The IBM NP4GS3 Network Processor provides multi-port interfaces for 10/100 Ethernet, Gigabit Ethernet and Packet over Sonet (POS). In particular the Ethernet configurations range from a 40-port 10/100 Ethernet

base design to a 1024-port 10/100 Ethernet product using a vendor switch fabric. Thus while the former supports 4 Gbit of traffic, the latter, where it is possible to connect up to 64 NPs, supports OC-48 to OC-192 connections.

2.1.1 Basic building blocks

The IBM PowerNP NP4GS3 Network Processor has eight main blocks as illustrated in Fig. 2.1.

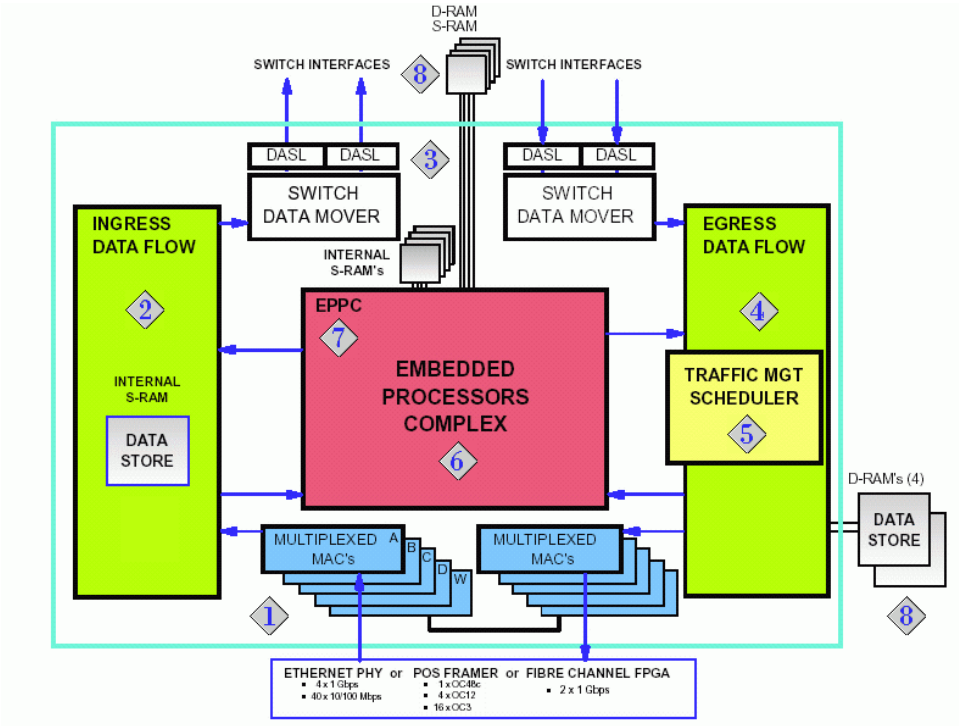

Figure 2.1: The NP basic blocks

1 - Ingress/Egress Physical MAC Multiplexor (PPM): this is the external interface, which receives and sends data packets. There are 5 Data Mover Units (DMU) for the ingress and 5 for the egress in each NP. Of these 5 pairs, 4 pairs are used to send/receive packets and the fifth is used for internal system communication. The DMUs are configured to provide the various combinations of physical interface support (10/100 FDX Ethernet, GBit, POS).

2 - Ingress En-queue/De-queue Scheduler (I-EDS): it starts processing the Inbound packets / frames. Filtering is done here. See Section 2.3.

3 - Switch Interface (SWI): it converts frames into cells via hardware and vice versa so that they can be transported across the SWI on their way to the appropriate Egress NP. Because the conversion is non-blocking and IBM's data aligned serial link (DASL) interface is used the bandwidth can reach 3.5 - 4.0 Gbps.

4 - Egress En-queue/De-queue Scheduler (E-EDS): it provides further packet processing/filter and traffic shaping. See Section 2.3

5 - Traffic Management Scheduler: it is an optional, configured NP component and is used to 'shape' or manage bandwidth on a per-flow basis.

6 - Embedded Processor Complex (EPC): this is where the 8 Dyadic Picocode Processor units (DPPUs) reside along with the Hardware Assisted Functions and the 9 Coprocessors. This system executes the picocode that provides the steady-state processing for packets. For example forwarding decisions are made here.

7 - Embedded PowerPC Complex (ePPC): it is a specialized PowerPC-405 processor, which can be used to provide CP functions.

8 - Storage areas throughout the system: there are three types of memory: internal static SRAMs, external DDR sDRAM(Double Data Rate Synchronous DDRAM) and ZBT SRAM (Zero Bus Turnaround). A large part of them is used for storing large forwarding tables.

2.1.2 The Embedded Processor Complex (EPC)

The EPC executes the picocode in parallel in 8 DPPUs . Each DPPU contains two non-preemptive, event-driven processor called Core Language Processors (CLPs) that share 8 Coprocessors, one coprocessor command bus and a memory pool, and each CLP is able to execute two threads. Thus the 8 DPPUs execute 32 threads in parallel. 28 threads, called General Data

Handlers (GDHs), are dedicated to frame processing and forwarding, and can be configured such that all handle their own unique frame and data frames. The remaining threads are used for building tables, processing messages from and to the ePPC, and for the in-band control mechanism between the EPC and all the other devices. This means that the NP can process 32 packets in parallel. The DDPU resembles the main CPU processor of a PC, and consequently the coprocessors acts as floating-point processor, input/output processor, video processor or other auxiliary processors.

A crucial basic block of the EPC is the Hardware Classifier (HC), which is able to parse frames on the fly and prepare them for processing by the picocode. In fact it provides a picocode label (also called entry point) which is the address of the first instruction to be executed. The parsing is accomplished based either on pre-defined masks (for example unicast/multicast, Ethernet/PPP, IP/IPX, etc) or on software-installed masks (i.e. for redirecting all packets from a specific IP source), which are stored in the Common Instruction Address (CIA) table.

PowerNP picocode

Picocode is an assembler-level programming language designed for the EPC's General Purpose Processors (GPPs), which contain Array Registers, Scalar Registers and General Purpose Registers (GPR). The GPRs, as shown in

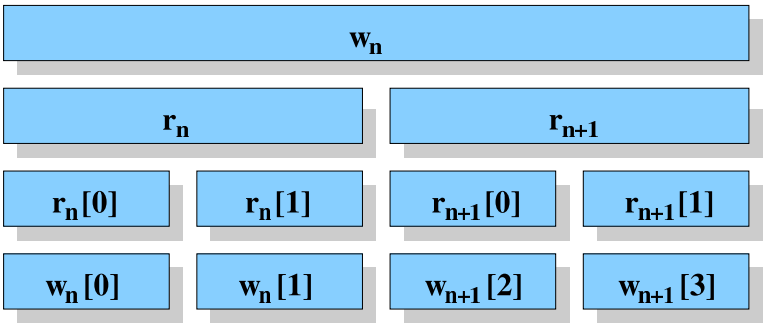

Figure 2.2: An example of a register view

Fig. 2.2, can be viewed as either sixteen 32-bit registers $(w_0, w_2, \ldots, w_{28}, w_{30})$ or thirty-two 16-bit registers $((r_0, r_1), \ldots, (r_{30}, r_{31}))$ or sixty-four 8-bit reg-

isters $((r_0[0], r_0[1], r_1[0], r_1[1]), \ldots, (r_{30}[0], r_{30}[1], r_{31}[0], r_{31}[1]))$. The CLP is a Big-Endian coprocessor. When a register is written to an array, the most significant byte is written first (see Fig. 2.3). Moreover, loading the constant `0xABCDEF` into a 32-bit register means buffering `0x00ABCDEF` for example.

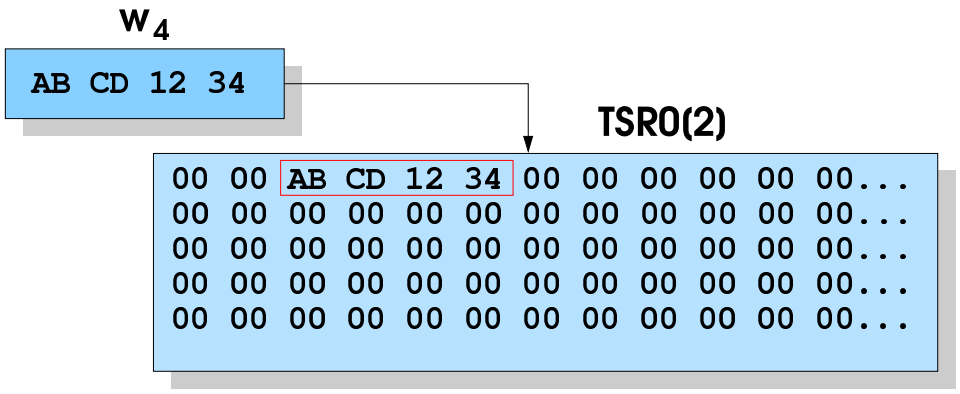

Figure 2.3: Big-Endian example

As we said, Picocode is an assembler-level language. A great part of the picocode operands has the common syntax:

```
operand destination, source
```

Let us show some examples of the load instruction: `ldr`. Data can be moved inside the GPRs and to or from the Coprocessor registers. We can move blocks of 16 or 32 bits, but we can also load 16-bit immediate values such as a constant into a register. For example,

```
ldr w4, w6
ldr r8, r4
```

copies all the bits of w_6 into w_4 and then the upper part into r_8, that is the upper part of register w_8. On the contrary, if we need to load a 32-bit immediate value (i.e. `\#0xABCD1234`) into a register then we are forced to use two instructions loading 16 bits each one. It is also possible to load bytes into/from an array (i.e Tree Search Register 2 - TSR2) for communicating with the coprocessors. The offset, indicating the point where we start reading or writing (we use `ldr` or `str`), can be fixed or a register value as shown in the next two instructions:

```
ldr TSR0[5], r4
str w2, TSR3[r0 + 5]
```

Moreover, using the `mov` instruction, it is also possible to write data in specific registers which are used only by the coprocessors (i.e. LCBA0/LCBA1 used by the TSE).

Particular versions of the `ldr` provide other functionalities such as "load with sign", "load and fill with zeros", etc. However the picocode set of instruction is not as extensive as the Assembler set. Table 2.1 presents a summary of the more common operands. In contrast, picocode also supports some high-level pre-processed constructs such as `@WHILE ... @ENDWHILE`, `@REPEAT ... @UNTIL`, `@IF ... {{@ELSE_IF ...} @ELSE...} @ENDIF`, non-recursive macro definitions (`@MACRO`) and aliases (`@EQU`) of registers or immediate values. However, for decreasing the number of CPU-cycles of the code, we avoided the use of `@IF` or `@WHILE` constructs and coded them directly using branches.

Table 2.1: List of picocode operands

arithmetic	add { c, i, s, ci, si, cs, csi }, sub { c }, cmp { s }
logical	and, or, not, xor, tst
register	ldr { h, hs }, ldrs { h, hs }, str
shift bits	sll, slr, asr, rotr
bit level	bitclr, bitset, bitxor
branch	b { z, nz, g, e, ne }, bal
moving data	mov, movm
coprocessor	cpx, wait { p, ok, okp, ko, kop }, key, keyc
other	ret, nop

Finally let us describe the ALU conditions code. Without needing branches for executing a block of code, we can execute several opcode instructions based on the result of a single test that modifies the ALU flags (zero flag, carry flag, sign flag, overflow flag). For example, the code lines

```
cmp w4, w2
ldr <ge> w6, w4 ; greater or equal
```

```
ldr <l>  w6, w2 ; less than
```

store the maximum between w_4 and w_2 into w_6. However, we must take care that the conditional instructions do not update the flags.

Hardware Assist Coprocessors

The hardware Assist Coprocessors are also integrated in the EPC. They run in parallel, in synchronous or asynchronous mode, with the engine executing the picocode. The parameters are passed through the scalar arrays (such as TSR (Three Search Register) or LCBA0, which we have already described), which are maintained per thread. Below we present a general description of all the coprocessors:

Checksum calculates and verifies frame-header checksums based on RFC 1071;

CAB interface is used for providing all DPPUs with access to internal registers, counters and memory for debugging or statistic gathering;

counter updates counters, i.e., the number of frames forwarded

Data Store provides access to the Ingress and Egress Data Stores. At most, 128 bits could be transferred per data transfer instruction;

Enqueue used for enqueueing frames to the switch and to the target port of the output queue;

Policy determines whether the incoming streams comply with four management algorithms specified in IETF RFCs 2697 and 2698;

String copy accelerates the data movement between coprocessor and memory pool;

Tree Search Engine (TSE) performs analysis through Fixed Match, Longest Prefix, or Software-controlled on tree searches. TSE uses an hash coprocessor to create the tree structure and two coprocessor locations

(LCBA0 and LCBA1), so that a thread can execute two searches simultaneously (TSE0 and TSE1);

Semaphore Manager is used for assisting in the sharing of resources such as tables or control structures using semaphores.

Let us now provide more details on the *hash coprocessor* which is integrated in the TSE. It implements some geometric hash functions yielding lower collision rates than conventional bit scrambling methods do, providing faster lookup and more powerful search engines. In fact data are not buffered in one large tree but in several smaller ones. The search-key is used as input of the hash coprocessor, and the result points to a specific small tree. However we can use the hash coprocessor as a stand-alone unit and compute only the hash scores. This black box accepts at most 192 bits, but it is possible to configure the unit to accept a smaller number. In Subsection 3.2.8, we will analyze the spreading of the first 32 bits.

The operand to invoke the hash coprocessor in an asynchronous way (`tse1_hka` for TSE1) accepts only one immediate value in which, at different bit positions, the user must specify the parameters. They are, mainly, the type of the hash function and the location of the input/output (the array where the hash key is and the array where the hash score will be). As all asynchronous commands, we also need an instruction to stop the main flow when we really need the result (i.e. `wait COP_tse1`).

The Control Point

While the NP performs all *steady-state functions* such as frame forwarding, L2-L3-L4 and higher frame processing, the Control Point executes all *non-steady-state functions* such as all functions that require special processing and logic. Common CP operations are updating and maintaining the OSPF, RIP or BGP databases, parsing IP packets with options, management, route discovery, route tree updates, and executing special policies designed by the vendors. One CP can control up to 16 NPs, and can be internal or external

to the NP.

All traffic at the NP that requires special logic is forwarded to the CP using a special frame type called Guided Frame. Conversely also the CP sends back guided frames to the NP. These frames encapsulate the real frame and may contain data or one or more commands. In this way the CP can for example install new routes in the forwarding tables (see Subsection 2.4.1).

The design of an algorithm should always be split in these parts: steady state and not-steady state. For this reason the CP will handle all *long-living flows* and associated timers in our load-balancing algorithm whereas the NP will handle only the forwarding of all normal streams (see Subsection 3.4.1).

In the Fig. 2.1 we observe that the CP operations are performed in the ePPC block, while the NP core is in the EPC. Between them there is a sort of mailbox. The NP and CP are connected to it by a PCI bus. They check the mailbox whether there are any new guided frames for them and if so they get them. Sending a guided frame is equivalent to sending data to the mailbox. The mailbox has a third PCI connection: it is for an external host processor running the CP software. In this way if the user wants to connect an external CP (it could be a UNIX workstation for example), he or she merely has just to plug the connector into the PCI socket of the NP, and the NP will never know that it is communicating with an external device.

2.2 Static and Dynamic Memory Management

The PowerNP has a different data memory organization than most other microprocessors on the market today. To access the memory, the NP must know not only the memory address but also the memory shape. For this reasons the block of memory is described by its *width* and *height*. Even if this is transparent to the user, these parameters specifies how the data are packed in the memory. Height describes how many addresses in memory the block uses while width describes how many memory banks it spans. The allowed configurations <width, height> are limited and depend on the

type of the memory banks (speed) and on the byte dimension of the block (structure). Fig. 2.4 shows how data are packed with $< 0, 3 >$ and $< 3, 0 >$ configurations. Note, looking at Table 2.2, we see that in NP syntax, the pair $< 1, 4 >$ is coded as $< 0, 3 >$.

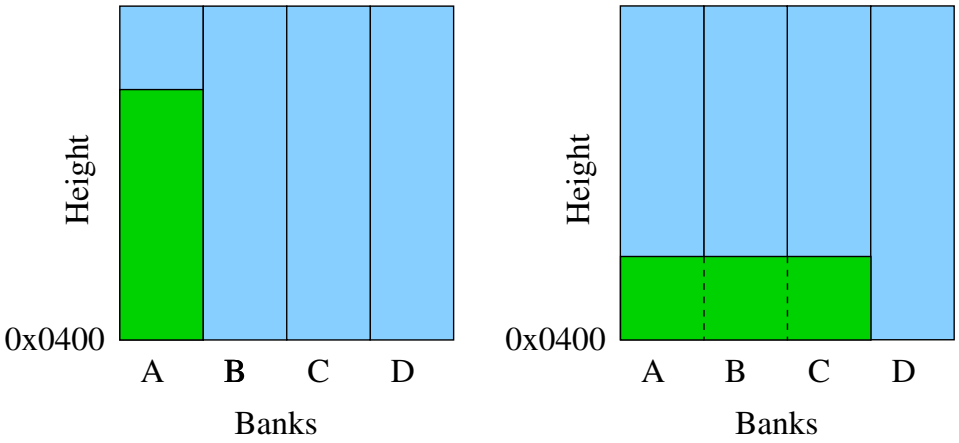

Figure 2.4: Memory block placement

In the first case a memory block at address 0x0400 will use addresses 0x0400 and 0x0401 and 0x0402, and span one memory bank, whereas in the second case it will use only one memory address, 0x0400, but it will span three memory banks. Table 2.2 shows some possible configurations for the internal RAM, which is the fastest.

Table 2.2: Some memory configurations

Memory Bank	Height	Width	Bit-element dimension	Offset
H0	1	1	128	1
H0	2	1	256	2
H0	4	1	512	4
H1	4	1	128	4
H1	6	1	192	8
H1	8	1	256	8

The user must specify how many elements there are in the table and <width, height> define the element dimensions such as the number of bytes that the NP is able to get from the memory in one access. To access to the next element of the table, the user must add the offset to the table address.

The dynamic memory is handled by using two tables. The first static table contains the size and shape of the next table and a pointer to its memory address. In this way,we can re-size the second table (which contains the real information) by changing the parameters in the first table. Thus, this kind of tables needs two memory accesses, and consequently they are slower even if we use internal memory.

2.3 Packet forwarding

We have already described all the basic blocks of the NP and thus we can explain better what happens when a packet arrives at the NP (Fig. 2.5).

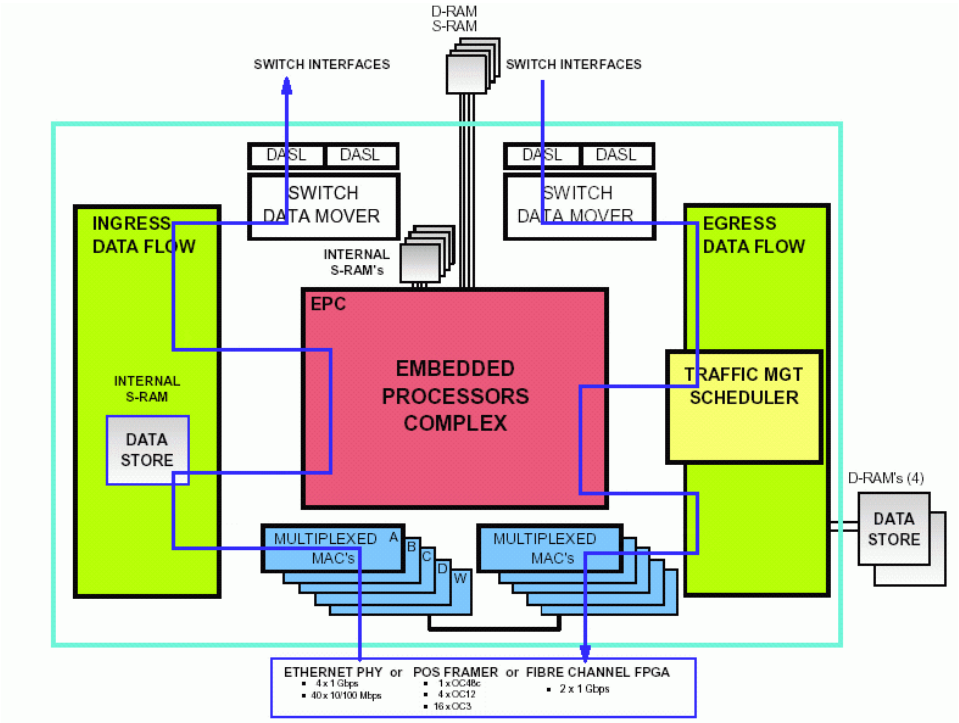

Figure 2.5: Basic packet flow inside the NP

Fig. 2.6 shows a typical configuration of two NPs (A and B) with a switch fabric used as a crossbar.

Ingress side The I-PPM receives a frame from the physical-layer device and forwards it to the I-EDS. The frame is immediately enqueued to the

EPC, where the Dispatch Unit fetches a portion of the frame and sends it to the next available thread. Simultaneously the HC determines the entry point based on predefined masks and sets some bits (i.e. unicast/multicast, L2 and L3 protocols) useful for picocode processing. The HC also decides, based on information stored in the CIA table, the Working Byte Count (WBC) i.e. the number of bytes of the frame which are available at the Ingress Data Store (IDS). This number is usually 64 bytes (Ethernet_header + IP_header_no_options + TCP_header $<$ 64). All this information is appended at the top of frame in a special header called Ingress Frame Control Block (I-FCB).

When the picocode starts to be executed the IDS memory is copied into the Data Pool memory that is a per-thread-reserved area. Let us focus now only on L3 operations for normal unicast IP packets. Based on the destination IP address an asynchronous tree search is started to obtain the forwarding port. The result provides several pieces of information. The most important are

FHF (Frame Header Format), which is a 4-bit value used by the HC on the egress side to determine the starting instruction address for that frame.

TB (Target Blade), which is the address of the Egress NP. Special values are also used for multicast. We will use always one NP, i.e. TB = 0.

TP (Target Port), which is the interface at the egress side to which the packet has to be forwarded.

This parameters are all inserted in the Egress Frame Control Block (E-FCB) which is the new header, replacing the I-FCB. As we can see in Fig. 2.6, this special header is used to pass the parameters from the ingress NP to the egress one. Another important parameter is the Frame Header Extension (FHE), which is a 32-bit free value defined by the user and used to pass parameters from the Ingress to the egress side.

The last operations of the ingress side are TTL and checksum verification, frame counter updates and other possible lookup due to other protocols (i.e. MPLS or BGP). Then the frame is enqueued and sent to the SWI.

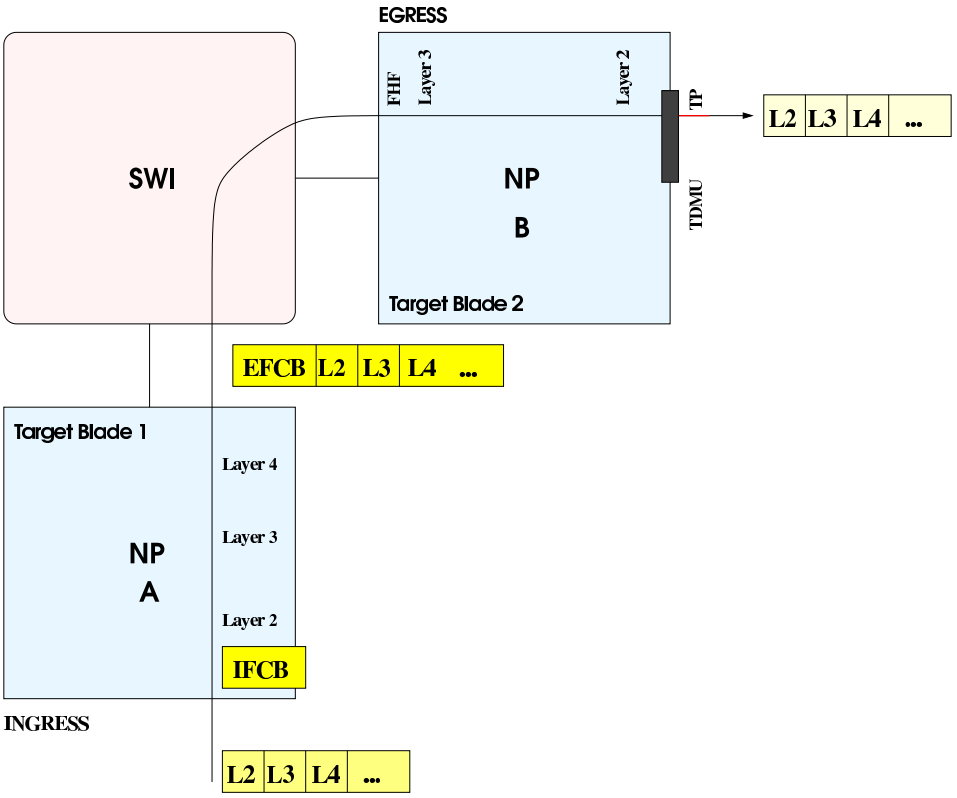

Figure 2.6: Typical device configuration

Egress side From the SWI interface of the NP matching the TB parameter, the frame is enqueued to its E-EDS. Here the Dispatch Unit fetches a portion of the frame (64 bytes typically) in the data pool memory and the HC, based on the FHF parameter, computes the picocode starting point. Other analyses could be done here such as the ARP lookup for resolving the MAC of the destination address, L2 alterations due to different L2 protocols, or physical synchronization. Finally the frame is enqueued to a specific queue/port (TP) of the E-PPM, and sent to the next hop.

Lastly, let us provide more details of the Dispatch Unit (see Fig. 2.7). We have already said that it copies a part of the frame from the Ingress

Data Memory (IDM) in the data pool memory, which is an area accessible to the picocode. At the egress side the frame is copied from the switch to the Egress Data Memory (EDM) and then back to the data pool memory. For these reasons, changes in the data pool area do not influence the real frame because it is only a mirror of the IDM/EDM. However, it is possible to load one or more quadwords (16 bytes) of the data pool into the IDM/EDM using the `IDIRTY/EDIRTY` commands. The `xDIRTY` commands accept only one parameter that specifies the block of quadwords to be updated. The `IDIRTY`

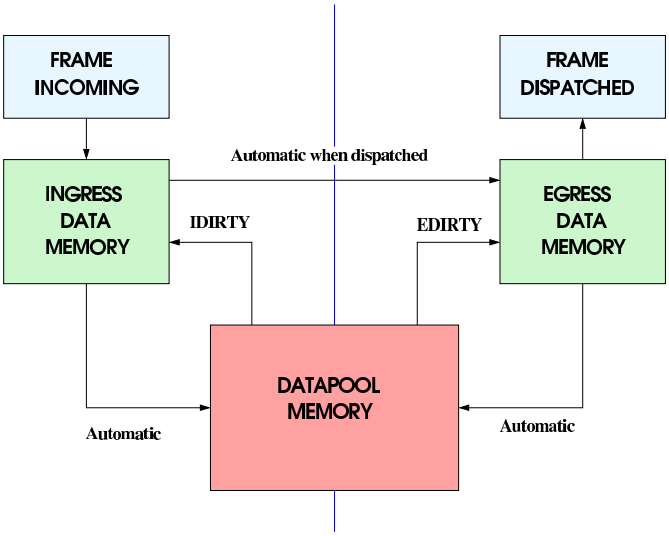

Figure 2.7: Relationship among IDM, Data pool and EDM

command has been really used in our algorithm. As shown in Fig. 3.34, it can happen that, at the ingress side, the NP must send the packet and other data to the CP. The data will be buffered, using `IDIRTY` command, in the source-MAC field of the packet.

2.4 Development Tools

2.4.1 Advanced Software Offering (ASO)

The ASO is a set of functionalities created for the customers that extends the Base Software Offering (that is the basic set of functionalities), and provides more functionalities for an easy and fast development of existing or

new protocols. All functionalities are written in picocode or, for the control software, in C/C++. The Base Software Offering provides basic services, for example, for managing the NP, for the guided frames, for the physical layers, table management, and NP booting. The list below shows some new functionalities developed in the ASO.

- NP diagnostic, deep management with MIB support for IPv4 or MPLS.
- Picocode and guided-frames control.
- Layer-2 functionalities such as bridging, spanning tree, layer filtering, PPP, and Ethernet protocols.
- Layer 3-4 functionalities, such as IP unicast/multicast filtering and forwarding, MPLS, diffserv, fragmentation, QoS, threshold load balancing, and the most common protocols such as TCP.

2.4.2 NPscope and NPsim

The NPscope (Fig. 2.8) is a debugger program used to test, monitor, and write picocode. It emulates the IBM NP4GS3 chip and provides all details about the execution (CPU cycles, execution time, etc).

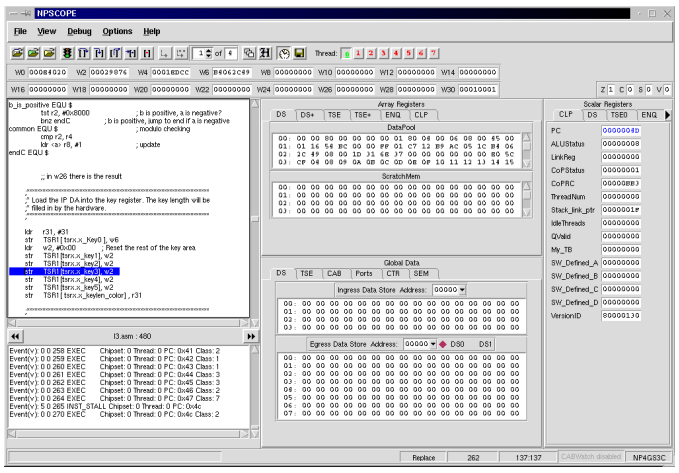

Figure 2.8: The NPscope

The NPsim emulates not only all functionalities of the NP but also, using TCL scripts, C/C++ code and the Linux Kernel, i.e. the functionalities of

the CP.

By simultaneously using these two tools and programs for sending/receiving real UDP/IP packets, it is possible to check not only the content of all GPRs, the tree structures, and all coprocessor states when the packet arrives, but also the consistency of the algorithm. In this way we can check, for example, the forwarding, and obtain some real results.

With NPSim we can also simulate the entire network topology. For example, using this *NETS* file we can describe the topology shown in Fig. 4.1:

```
# ID@loop_back     NP_ID:interface
 120@127.0.0.1:2   250:2
 130@127.0.0.1:3   250:3
 140@127.0.0.1:4   250:4
 150@127.0.0.1:5   250:5
# NP receives packets on interface 21 from ID 100 250:21
100@127.0.0.1:21
```

The NPSim simulates different IP addresses on different interfaces using only the loop-back address, the node-ID number and the NP-interface. But the entire process is transparent to the Kernel which is extended with these virtual interfaces. For these reasons, the C-application that we wrote for sending UDP packets (*pktsend*) requires the ID of the node which is sending this packet. For example, this command line simulates a host (ID 100) sending an UDP packet (with a payload of 40 bytes and having IP destination 12.0.0.2 and IP source 137.91.24.41) to interface 21 of the NP:

```
./pktsend -v -sim 250:21 clear 12.0.0.2 40 137 91 24 41
```

In the same way, the receiving C-application (*pktrecv*) requires the ID of the node to the which the UDP-dumper must be attached.

Chapter 3

Designing the load-balancing algorithm

In this chapter we will lay the theoretical foundation for some of the concepts used in this thesis, and show several possible solutions fitting our needs. We will discuss how the NP processes all incoming packets and how it decides to forward the packets and to distribute the flows across different outgoing links.

The chapter is organized in four sections. In the first section we will discuss the Robust Hash Mapping, which is a basic block for creating high-performance forwarding functions. Unfortunately this solution cannot be implemented as is on the PowerNP, and we will emphasize the main disadvantages. In the second section we will present our mapping algorithm, derived from the Robust Hash Mapping, which does not present these disadvantages. However for implementing this new algorithm, we had to solve some new specific challenges, such as generating random values having negative exponential distribution. We will provide the details about all possible and feasible solutions. In the third section we will describe the control loop used to monitor system utilization as well as to provide best performance, also under heavy traffic conditions. We will test the performance of the control loop together with our packet-forwarding algorithm using some java simulations. In the fourth and last section, we will present the IBM proposal for handling TCP flows. After the description of the NP-state machine we will show the final algorithm that will be implemented on the NP.

3.1 The forwarding algorithm

3.1.1 Requirements

Let us suppose that the N real servers connected to the NP have the same IP address to avoid NAT translation from the load-balancer address to those of the real servers (see Subsection 4.1.1 for more details about the addressing). Let us assume a packet w (we are interested only in its header part) arriving at the NP. 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 our specific forwarding algorithm. In this case we need a mapping function $f(\cdot)$ that, using some packet fields $\vec{v}$ as input, returns a number $j \in [1, N]$ that denotes the outgoing port to which the packet w will be forwarded. Note that this function must be deterministic in order to guarantee that packets belonging to the same flow will all be forwarded to the same outgoing port. The function must also guarantee good load balancing without using table or state information (stateless). The function must also guarantee heterogeneous load distribution over ports (meaning a non-uniformly load distribution over the N ports) for many reasons. For example, either because the user wants to spread the bandwidth unequally over the real servers (so that the fastest ones receive more load) or because the CP increases or decreases the load of a specific port during the control-loop phase (in order to guarantee the best system utilization). Accordingly, we set some (dynamic) load percentage p_j for each port j.

Nevertheless, when the CP changes those percentages, some ongoing flows (suppose TCP) could be redirected to another real server. In Section 3.4 we will call these redirected connections *long-living flows*. We will see that if the number of those flows is too large, the NP/CP will be not able to handle the totality of them. Some connections will certainly be dropped, and thus the users will notice connection delays. We need to avoid problems of this kind, and thus we need to guarantee a general minimum flow disruption property: the number of flows redirected in that case should be minimal.

Finally, fault tolerance is provided not only by the NP's robustness but also by the control loop, which assures that even if a real server fails, the entire system is able to adjust to the failure quickly and gracefully, i.e. with minimal flow disruption.

3.1.2 Robust Hash Mapping

Robust Hash Mapping has been introduced to solve the disruption problems when, for example, real servers are added or removed from the server farm. Using an example we will show the flaws of basic mapping functions and conversely the advantages provided by the Robust Hash Mapping.

Let $h(\cdot)$ be a hash function that maps the set of all n-bit binary numbers to a hash space H. This hash space H is then partitioned into N sets and we establish a one-to-one correspondence between real server and set. Lastly let us call $\vec{v}$ the set of predefined packet fields that do not change within a particular flow. Each v_i represents a piece of data within the packet for example the IP addresses or the port numbers. When a packet w arrives at the NP, the NP extracts $\vec{v}$, computes $h(\vec{v})$ and obtains j, i.e. the j-th outgoing link or j-th interface. This is the simplest Hash-Mapping scheme. But as explained in Ref. [13], it has a critical flaw. Whenever a real server is being added or removed, the fraction of flows that will change their real server can be large. This is due to the change of the thresholds in the hash space. Thaler and Ravishankar [14] refer to this fraction as the *disruption coefficient*. To understand better what happens when a new Real Server is added, let us look at Fig. 3.1.

Initially, the hash space is divided into N sets:

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

If the hash function produces the score $sc \in \left[\frac{j}{N}, \frac{j+1}{N}\right]$, then port j will be chosen. Let us now suppose that all sets have the same probability, i.e. sc is uniform over the H space. Let us suppose that a new Real Server comes

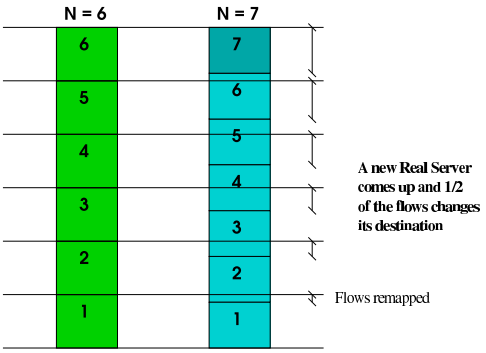

Figure 3.1: Minimal disruption problem in some mapping functions

up. After that, the hash space will be divided as follows:

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

As we can easily see only the flows in the following subsets have not changed their real server destination; namely

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

We can also easily compute the sum of these intervals to evaluate the disruption coefficient:

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

This means that in this case 50% of the flows have changed destination. Thus this family of mapping functions $h(\vec{v})$ is not a good candidate for our needs. The number of flows, that change their destination server following the changes in the sets into which the H space is divided, is too large, and we have already explained why we want to avoid these kinds of problems. Our aim is that even if there are changes in the H space (due to server failures or triggered by the CP control loop), the number of flows involved has to be minimal.

The Robust Hashing Mapping solves the problem of hashing functions with flow-disruption problems. With the *robust hash mapping* (see Ref. [13]), the real-server name (in our case the port number) and $\vec{v}$ are used together to generate a hash value or score. Thus for every link j we will compute

$h(\vec{v}, j)$, and the final outgoing port will be the one with the maximum score. More specifically

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ h(\vec{v}, j) &= \max_{k \in [1,N]} h(\overrightarrow{v}, k). \end{aligned} \tag{3-1}$$

Now, Ross [13] demonstrates that if, for example, we try to add a new Real Server, only a fraction $\frac{1}{N+1}$ of the flows will reside on the wrong side (see Fig. 3.2).

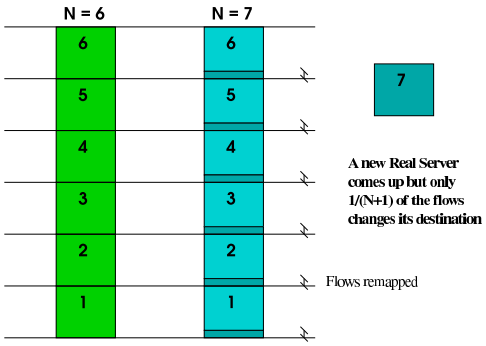

Figure 3.2: Robust hash mapping avoids the flow-disruption problem

3.1.3 Multiplier theorem

The robust-hash method, as described above, guarantees only uniform load balancing, which is not sufficient. Ross [13] introduced weights to (3-1) to provide different load distributions over different links (p_j) and consequently to guarantee *heterogeneous load distribution over ports*. Using these weights as multipliers of the hash scores, Ross. [13] proposed a Robust Hash Mapping with a weighted forwarding decision:

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ x_j \cdot h(\vec{v}, j) &= \max_{k \in [1,N]} x_k \cdot h(\vec{v}, k) \end{aligned} \tag{3-2}$$

Ross. [13] proposed and proved a theorem for computing the weights x_i corresponding to the load percentages p_i with linear complexity ($\sim O(N)$).

This is the statement:

Theorem 1 (Keith W. Ross). Let $p_1, \ldots, p_N$ be given target probabilities. Reorder the caches 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:

$$x_n = \left[\frac{(N-n+1)(p_n - p_{n-1})}{\prod_{i=1}^{n-1} x_i} + x_{n-1}^{N-n+1} \right]^{\frac{1}{N-n+1}}. \tag{3-3}$$

Then the robust hash algorithm with multiplier $x_1, \ldots, x_N$ will route the fraction p_n of incoming flows to the n-th real server for $n = 1, \ldots, N$.

Drawbacks The theorem assumes that the probability distribution of the hashed values is $U(0,1)$. This means that using packet information we must produce uniformly distributed values which have to be multiplied by the weights.

This has several drawbacks because picocode does not have a specific operand for the multiplication. For example, 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 (3-2) is $\sim O(\text{const} \cdot 16 \cdot N)$.

Knowing these limitations, we tried to implement a very fast multiplication in picocode to evaluate the actual performance in terms of cycles. This is the abstracted algorithm of the picocode solution:

```
      Result = 0
start:
      j = FIND_FIRST_MSB_IN_MULTIPLIER
      IF (j != Error) {
          multiplicand = multiplicand << j
          Result = Result + multiplicand
          set_to_0 bit j multiplier
          multiplicand = multiplicand >> j
          GO TO start
      }
```

All these instructions are referred to NP4GS3 opcodes. Note that in this case the jump instruction, which in general is very expensive in terms of cycles, requires only one cycle each time because the code fits the size of buffer in which the last instructions are cached. Thus the processor requires only 1 cycle for translating the `start` label into an offset for the program counter. However, simulations of this algorithm in the NP4GS3 show that the number of cycles only depends, as expected, on the number of bits '1' in the multiplier and not, for example, on the position of these bits or on the multiplicand. If we suppose that the multiplier could uniformly assume values in the interval $[0, 2^{16} - 1]$, we can assume that the numbers with eight bits equal to '1' represent the cycle average. The NP4GS3 standard forwarding algorithm (as described in Chapter 2) needs about 800 cycles to process and forward an IP packet. Results shows that a 16-bit multiplication needs about 165 cycles. This multiplication should be executed N times for every packet, and this clearly is too costly.

Finally if we need to increase the number precision, say that we need numbers with 32 bits, then the results will require a 64-bit register, which is not available in the NP4GS3 architecture, and thus the multiplication is not feasible if we need to use a 32-bit representation for our weights. Unfortunately as we will show in Subsection 3.2.2, we need at least 10 digits for the integer part. The remaining 6 digits for the fractional part do not assure enough precision in the weight representation, and thus we can conclude that the solution as presented in (3-2) is not feasible.

3.2 A feasible mapping algorithm

We have seen that the mapping algorithm (3-2) cannot be implemented because of the absence of a multiplication operand. Thus, our first idea was

to find a new mapping algorithm using another operand:

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ x_j \star h(\vec{v}, j) &= \max_{k \in [1,N]} x_k \star h(\vec{v}, k) \end{aligned} \tag{3-4}$$

However all attempts to replace the '$\star$' operand with other operands such as '+', '-', '$\oplus$', or '>>' have been unfruitful. In fact even if we obtained a newer and simpler mapping algorithm, we would not be able to compute the weights from the load percentages because we had to solve non-linear systems of N equations. Moreover, when the control loop adapts the weights, it is impossible to verify the adaptation has a minimal disruption property. For these reasons, we concentrated our efforts on finding a simpler but equivalent formulation of (3-2).

3.2.1 Offset lemma

This lemma proposes a new way to compute the weights x_i that are now used as offsets. The idea is that by applying the logarithm, which is a monotonous increasing function, to both sides of (3-2) we maintain the equality.

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ \log(x_j \cdot h(\vec{v}, j)) &= \max_{k \in [1,N]} \log(x_k \cdot h(\vec{v}, k)) \end{aligned} \tag{3-5}$$

Moreover 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 can also note that the logarithm of uniformly distributed values produces an exponential distribution (in our case a negative exponential).

Now, let us fully demonstrate this lemma for a more generic case (for example with the sum of weights as parameter), and then we will show the correspondence between the weights computed with the lemma and those computed with Ross's theorem.

Lemma 1 (Riccardo Russo). Let $p_1, \ldots, p_N$ be given target probabilities. Reorder the caches so that $p_1 \leq \ldots \leq p_N$. Let

$$x_1 = \frac{\ln Np_1 + S}{N},$$

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

$$x_n = \frac{1}{N-n+1} \cdot \ln \left\{ \frac{(p_n - p_{n-1})(N-n+1)}{\exp\left(\sum_{i=1}^{n-1} x_i - S\right)} + \exp\left(x_{n-1} \cdot (N-n+1)\right) \right\} \tag{3-6}$$

Then the robust hash algorithm with offsets $x_1, \ldots, x_N$ for every $h_1, \ldots, h_N$ score will route the fraction p_n of incoming flows to the n-th Real Server for $n = 1, \ldots, N$

Proof - Let $x_1, \ldots, x_N$ be an arbitrary set of offsets satisfying $x_1 \leq \ldots \leq x_N$. Let $h_1, \ldots, h_N$ be the hash values associated with each of the N caches. Because the h_ns are outputs of a hash function, they can be taken to be independent, uniformly distributed random variables. We take each h_n to be distributed as follows:

$$f_{h_n}(x) = \begin{cases} \exp(x) & \text{if } x \leq 0 \\ 0 & \text{if } x > 0 \end{cases}.$$

Let $Z_n = x_n + h_n$ be the n-th summed hash value. Note that the Z_ns are independent and distributed as follows:

$$f_{Z_n}(x) = \begin{cases} \exp(x - x_n) & \text{if } x \leq x_n \\ 0 & \text{if } x > x_n \end{cases}.$$

Note also that

$$P(Z_n \leq x) = \begin{cases} \exp(x - x_n) & \text{if } x \leq x_n \\ 1 & \text{if } x > x_n \end{cases} \tag{3-7}$$

is equal to the PDF of Z_n. Let $Z_{(n)} = \max(Z_1, \ldots, Z_{n-1}, Z_{n+1}, \ldots, Z_N)$. Let q_n be the probability that n has the largest summed hash value, that is,

$q_n = P(Z_{(n)} \leq Z_n)$. Conditioning on $Z_n = x$, we obtain

$$\begin{aligned} q_n = P(Z_{(n)} \leq Z_n) &= \int_{-\infty}^{x_n} P(Z_{(n)} \leq x) \cdot \exp(x - x_n) dx \\ &= \int_{-\infty}^{x_n} \prod_{i \neq n} P(Z_i \leq x) \cdot \exp(x - x_n) dx \\ &= \int_{-\infty}^{x_n} \frac{\prod_{i=1}^{N} P(Z_i \leq x)}{P(Z_n \leq x)} \cdot \exp(x - x_n) dx \\ &= \int_{-\infty}^{x_n} \frac{\prod_{i=1}^{N} P(Z_i \leq x)}{\exp(x - x_n)} \cdot \exp(x - x_n) dx \\ &= \int_{-\infty}^{x_n} \prod_{i=1}^{N} P(Z_i \leq x) dx. \end{aligned} \tag{3-8}$$

Let us look at a possible configuration to understand how to split the integration interval into several parts (Fig. 3.3).

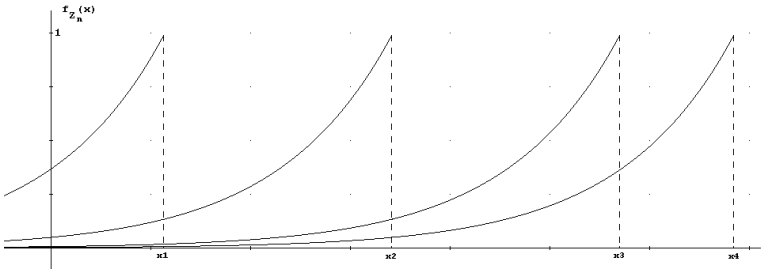

Figure 3.3: Weights example

If $n \neq 1$ we can split the interval $[-\infty, x_n]$ as follows:

$$\begin{aligned} [-\infty, x_n] &= [-\infty, x_1] + [x1, x_n] \\ &= [-\infty, x_1] + \sum_{j=1}^{n-1} [x_j, x_{j+1}] \end{aligned} \tag{3-9}$$

Using Eqs. (3-8) and (3-9) we obtain

$$q_n = \int_{-\infty}^{x_n} \prod_{i=1}^{N} P(Z_i \leq x) dx \tag{3-10}$$

$$= \int_{-\infty}^{x_1} \prod_{i=1}^{N} P(Z_i \leq x) dx + \sum_{j=1}^{n-1} \int_{x_j}^{x_{j+1}} \prod_{i=1}^{N} P(Z_i \leq x) dx \tag{3-11}$$

We must now get an explicit expression for the products in the above expression. For the first addend, where $-\infty < x \leq x_1$, we observe using (3-7),

that $P(Z_i \leq x) = \exp(x - x_i)\ \forall i = 1,\ \ldots,\ N$. For the second addend, where $x_j \leq x \leq x_j + 1$, we observe that

$$P(Z_i \leq x) = \begin{cases} 1 & \text{if } i \leq j \\ \exp(x - x_i) & \text{if } i > j \end{cases}.$$

Thus, Eq.(3-11) becomes

$$\begin{aligned}
q_n &= \int_{-\infty}^{x_1} \prod_{i=1}^{N} P(Z_i \leq x)dx + \sum_{j=1}^{n-1} \int_{x_j}^{x_{j+1}} \prod_{i=1}^{N} P(Z_i \leq x)dx \\
&= \int_{-\infty}^{x_1} \prod_{i=1}^{N} \exp(x - x_i)dx + \sum_{j=1}^{n-1} \int_{x_j}^{x_{j+1}} \prod_{i=j+1}^{N} \exp(x - x_i)dx \\
&= \int_{-\infty}^{x_1} \exp\left\{\sum_{i=1}^{N}(x - x_i)\right\}dx + \sum_{j=1}^{n-1} \int_{x_j}^{x_{j+1}} \exp\left\{\sum_{i=j+1}^{N}(x - x_i)\right\}dx \\
&= \int_{-\infty}^{x_1} \exp\left(Nx - S\right)dx + \sum_{j=1}^{n-1} \int_{x_j}^{x_{j+1}} \exp\left\{(N-j)x - \sum_{i=j+1}^{N} x_i\right\}dx \\
&= \frac{1}{N} \cdot \exp\left(Nx - S\right)\Big|_{-\infty}^{x_1} + \sum_{j=1}^{n-1} \int_{x_j}^{x_{j+1}} \exp\left\{(N-j)x - S + \sum_{i=1}^{j} x_i\right\}dx \\
&= \frac{1}{N} \cdot \exp\left(Nx_1 - S\right) + \sum_{j=1}^{n-1} \frac{1}{N-j} \exp\left\{(N-j)x - S + \sum_{i=1}^{j} x_i\right\}\Big|_{x_j}^{x_{j+1}} \\
&= \frac{1}{N} \cdot \exp\left(Nx_1 - S\right) + \\
&\quad + \sum_{j=1}^{n-1} \frac{\exp\left\{\sum_{i=1}^{j} x_i - S\right\}}{N-j} \cdot \left\{\exp\left[(N-j)x_{j+1}\right] - \exp\left[(N-j)x_j\right]\right\}.
\end{aligned} \tag{3-12}$$

Finally from Eq. (3-12) we have

$$\begin{aligned}
q_n &= q_{n-1} + \frac{\exp\left\{\sum_{i=1}^{n-1} x_i - S\right\}}{N-n+1} \cdot \\
&\quad \cdot \left\{\exp\left[(N-n+1)x_n\right] - \exp\left[(N-n+1)x_{n-1}\right]\right\}.
\end{aligned} \tag{3-13}$$

The desired result follows by setting $q_n = p_n$, $n = 2,\ \ldots,\ N$, and solving for x_n in Eq. (3-13). If $n = 1$ we only need to compute the first integral of

Eq. (3-11) and obtain

$$\begin{aligned} q_1 &= \int_{-\infty}^{x_1} \prod_{i=1}^{N} P(Z_i \leq x)dx \\ &= \int_{-\infty}^{x_1} \prod_{i=1}^{N} \exp(x - x_i)dx \\ &= \int_{-\infty}^{x_1} \exp\left\{\sum_{i=1}^{N}(x - x_i)\right\}dx \\ &= \int_{-\infty}^{x_1} \exp\left(Nx - S\right)dx \\ &= \frac{1}{N} \cdot \exp\left(Nx_1 - S\right). \end{aligned} \tag{3-14}$$

The p_1 result follows by solving for x_1 in Eq. (3-14).

Using this theorem we can easily compute the weights x_i (time complexity $\sim O(N)$) that are the offsets of our robust hash function. Thus the mapping algorithm becomes

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ x_j + h(\vec{v}, j) &= \max_{k \in [1,N]} x_k + h(\vec{v}, k). \end{aligned} \tag{3-15}$$

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

3.2.2 Details about the new mapping algorithm

The lemma (3-15) presents the statement of the multiplier theorem only in another way. It is not too difficult to see and prove that if x_i and x_i' are weights computed with (3-2) and (3-15), S is the sum of the weights x_i', and assuming that $\prod_{j=1}^{N} x_j = 1$, then

$$x_i' = \log x_i + S/N. \tag{3-16}$$

This is important because all the properties described in Ref. [13] and Ref. [15] are also true for the set of weights x_i'. For example, Ref. [15] introduced a control loop, which adapts the weights, having a minimal disruption

property. Because of the equivalence (3-16), the adaptation on the weights x'_i will maintain the same property.

Another important issue is understanding what kind of values the weights can have. For example we know that the x_i are always greater than zero. This is certainly not true for the 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 we need for representing the integer part in order to guarantee that almost all possible percentage combinations (and thus the corresponding weights) are represented. A C/Matlab simulation was created to display the weight distribution. For $N\_CYCLES = 10000$, we randomly assigned percentages to the $N = 8$ ports and then computed the weights with both mapping algorithms. Using two vectors with M cells each, we divided the space $[-L, L]$ into M intervals, and counted the occurrences X_i of the weights using the corresponding counters at position

$$\frac{L}{2} + round\left(\frac{X_i \cdot M}{2L}\right).$$

The sum of the weights obtained with (3-15) is 0. The results obtained are shown in Figs. 3.4 and 3.5.

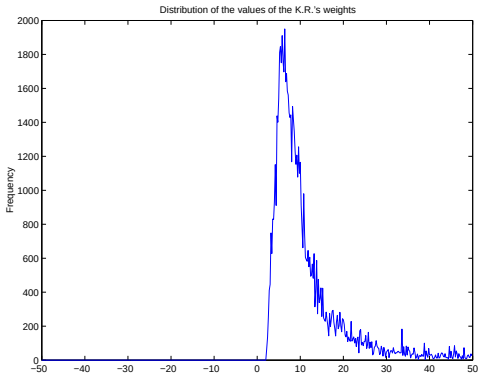

Figure 3.4: Ross's Weights distribution. L = 50, M = 500

The largest and smallest values of the weights (for (3-3) and (3-6) re-

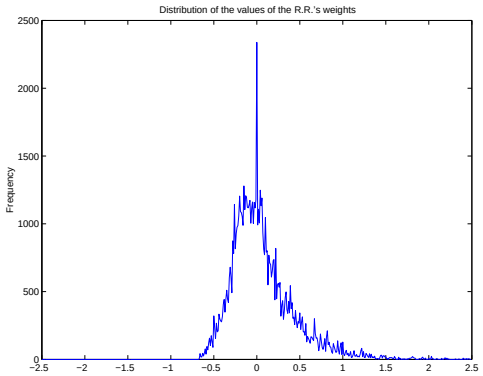

Figure 3.5: Russo' weights distribution. L = 2.5, M = 500

spectively) during the simulation were

$$\begin{cases} \max = 592.083951 \\ \min = 0.231048 \end{cases} \quad \text{and} \quad \begin{cases} \max = 6.383648 \\ \min = -1.465130 \end{cases}$$

Considering the two figures, it is easy to note that in the NP we need $1+3$ bits instead of $0+10$ to represent the integer part (and sign) of the weights described in (3-6).

3.2.3 Generating a negative exponential

The new mapping presented in (3-15) requires that the NP computes exponentially distributed scores using packet fields. We tried to look for a fast and easy solution for generating scores with this kind of distribution. Let us analyze them, trying to understanding whether they could be implemented in picocode.

- Ahrens and Dieter [16] developed a specific algorithm for 8086 assembler. Here is its main part:

  1. Generate a uniform value U and set $G \leftarrow$ c.
  2. Set $U \leftarrow U + U << 1$. If $U \geq 1$ go to 4.
  3. Set $G \leftarrow G + \ln 2$ and go to 2.
  4. Set $U \leftarrow U$ - 1. If $U > 1$ go to 6.
  5. Return $X \leftarrow G + A/(B - U)$.
  6. Generate U and set $Y \leftarrow a/(b - U)$.
  7. Generate U'. If $(U' + D)(b - U)^2 > \exp^{-(Y+c)}$ go to 6.

8. Return $X \leftarrow G + Y$

The main problem of this algorithm are the lines 5, 6 and 7. The authors propose a Taylor series for solving the problems at 7. Unfortunately the picocode do not have division and multiplication operands, and thus we cannot use this solution.

- Hamilton [17] increased the performance of the algorithm of Ref. [16] but the arithmetic problem remains.

- Marsaglia [18] proposed a new algorithm which uses rectangles to approximate the desired distributions. However, it is too difficult to implement in picocode.

- Fernández and Rivero [19] need two uniformly distributed variables (m and n) to generate the exponential ones (r_m and r_n). This is the main part of their algorithm:

  1. $m = (m + 1) \bmod N$
  2. n = IntegerUniformNumber(0, N-1)
  3. if $n = m$ go to 1
  4. x = RealUniformNumber(0, 1)
  5. $S = r_m + r_n$
  6. $r_m = xS$
  7. $r_n = (1 - x)S$

  As we can see the algorithm has a multiplication at line 6 and moreover the last exponential score is also used in the process to generate the next one. Thus, the result is not deterministic and consequently the same input does not generate the same output.

- Wallace [20] proposed a very fast, simple algorithm for generating two negative exponential variables (p^1, q^1) using only low-level arithmetic.

  (a) if $(p \leq q)$ $\{p^1 = 2p,\ q^1 = q - p\}$
  else $\{p^1 = 2q + 1,\ q^1 = p - q - 1\}$

(b) $s = p + q$
$p^1 = p - s/2$
if$(p^1 < 0)$ $p^1 = p + s + 1$
$q^1 = s - p^1$

(c) $s = p + q$
$p^1 = p - s/4$
if$(p^1 < 0)$ $p^1 = p^1 + s + 1$
$q^1 = s - p^1$

(d) Same as (b) save that, after s has been computed, odd-numbered binary digits of p are replaced by the corresponding digits of q.

One uniformly distributed variable is required for selecting, in each step, one out of four possible sequences of operations. After each step $p \leftarrow p^1$ and $q \leftarrow q^1$ and thus the subsequent values depend on the preceding ones. As in Ref. [19] the result is not deterministic;

- The acceptance-rejection method (Von Neumann, 1951) which however still needs multiplication and division operations.

3.2.4 The inverse transform method

Thus, no specific algorithm fits our needs well. The inverse transform method that generates a negative exponential variable (E) by merely computing the logarithm of uniformly distributed realizations (U) is the last of our known feasible solutions:

$$E = \log(U)$$

$$\text{with } f_E(x) = -e^x \text{ if } x < 0, 0 \text{ if } x \geq 0$$

How can we compute the logarithm? Two solutions are feasible:

- Put all values into a (static) table. Considering that for each port we need to look up in the table, this solution is quite expensive. Moreover we need to solve the collision problem. Because of the small size of the fastest memory we can store only a small table, say, that we load 256 8-bit values. This means that two different packets can generate the same uniformly distributed value. Even if the probability of a collision is in this case very high (say 2^{32} values mapped in 2^8 sets)

our Matlab simulations have shown that this does not influence the mapping algorithm too strongly. However we propose an alternative easy-to-code solution that assures that two different inputs produce two different hash scores.

Given two 32-numbers, $0\text{x}x_1x_2x_3x_4x_5x_7x_5x_8$ and $0\text{x}y_1y_2y_3y_4y_5y_7y_5y_8$, that produce the same hash score $0\text{x}z_1z_2000000$, and knowing that the logarithm is a monotonous increasing function we can compute the two numbers:

$$\begin{aligned} A &= 0\text{x}z_1z_2000000 + 0\text{x}00x_3x_4x_5x_6x_7x_8 \\ B &= 0\text{x}z_1z_2000000 + 0\text{x}00y_3y_4y_5y_6y_7y_8 \end{aligned}$$

Fig. 3.6 and Fig. 3.7 (which is an enlarged view) show how the proposed solution approximate the logarithm function. Notice that the proposed solution is equal to the logarithm function only in 256 points, whereas the approximation works quite well in the interval [0.15, 1].

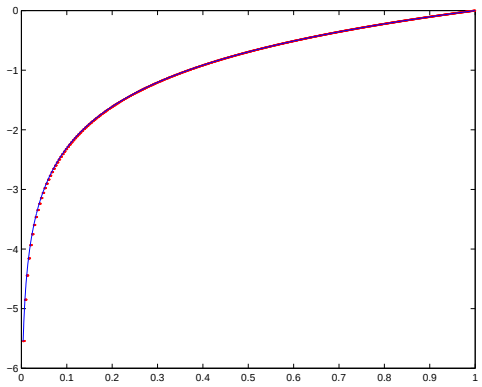

Figure 3.6: Comparison between the logarithm function and the table approximation

- Approximating the logarithm with the Taylor Theorem. Because we cannot execute the multiplication, we will try this approximation:

$$\log U = \log(1 - U') = -U' + o(U').$$

This approximation is really rough, but it has the special quality that it is really fast.

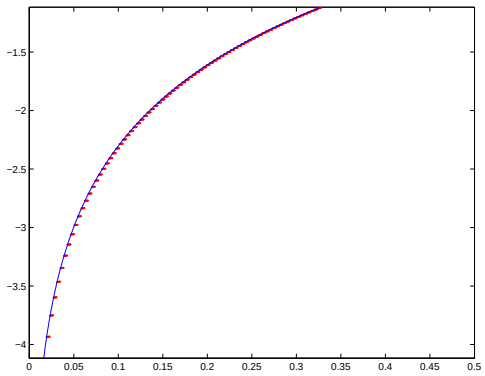

Figure 3.7: Detail of Fig. 3.6. No collision property

We chose to implement solution (3-15) using the Taylor approximation because it is really fast and easy to implement. In fact, as we can see, we have to subtract the hash scores from the weight, and this require only two CPU-cycles. The final mapping algorithm is the following:

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ x_j - h(\vec{v}, j) &= \max_{k \in [1,N]} x_k - h(\vec{v}, k), \end{aligned} \tag{3-17}$$

where x_j is computed using (3-6) and $h(\vec{v}, j)$ is a uniformly distributed pseudorandom function.

Using a Matlab simulation we tried to compare the various (feasible) solutions to determine what the precision of the mapping is. The user sets the percentages p_i, Matlab computes the weights x_i and generates *N_SAMPLES = 10000* random uniform numbers, as assumed in the theorem and in the lemma, which simulate the hashing random values. Then it executes (3-2), (3-15), (3-15) with a table approximation, and (3-17). Counters are used to buffer the number of packets sent to each link. Figures 3.8 and 3.9 show two simulation outputs.

As we can see the Taylor approximation is quite rough, but it is sufficient for us because it allows for a very fast execution.

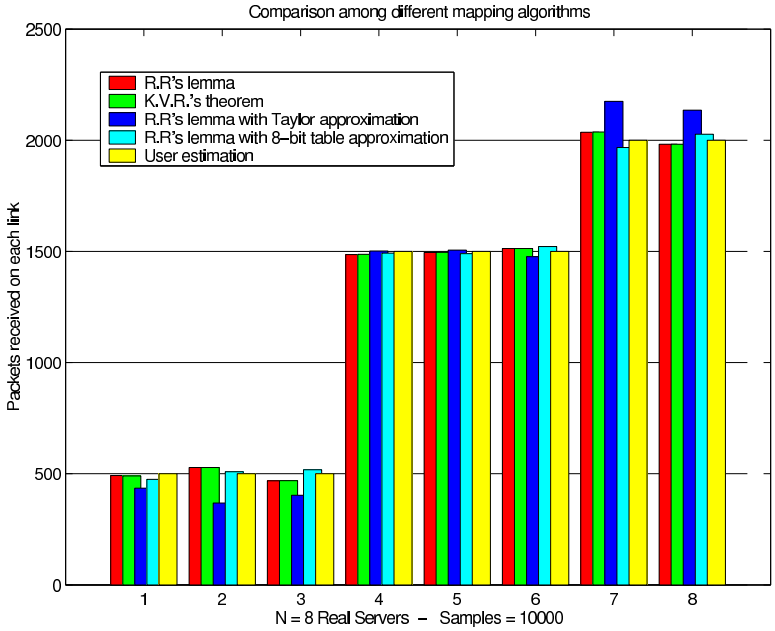

Figure 3.8: First example of the comparison of different mapping algorithms

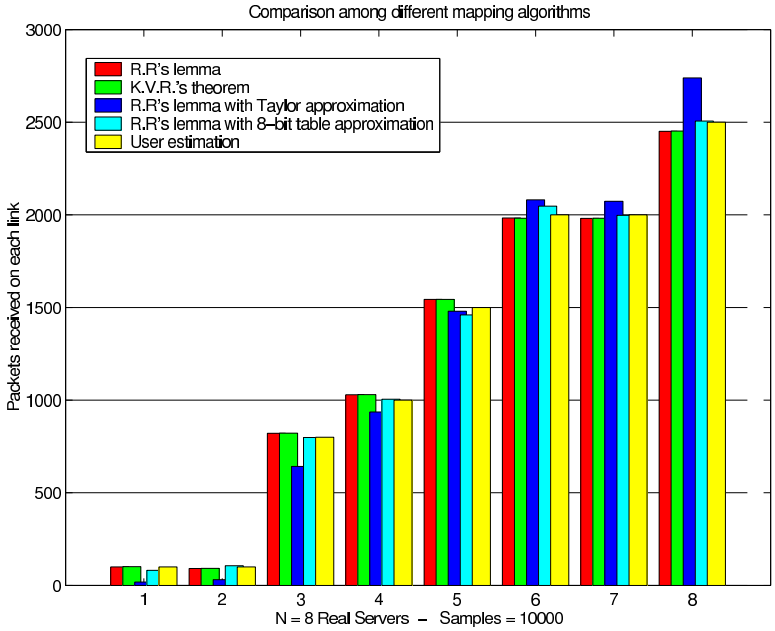

Figure 3.9: Second example of the comparison of different mapping algorithms

3.2.5 Pseudorandom functions with uniform distribution

The mapping algorithm (3-15) requires hash scores with a negative exponential distribution. We have seen that no specific algorithm fits our needs, and that the only feasible way for generating the desired distribution is using the

inverse transform method that computes the logarithm of a uniformly distributed variable. The logarithm was approximated using Taylor Theorem and in this paragraph we will present all the feasible solutions for generating uniformly distributed hash scores.

In the Subsection 3.2.3 we emphasized that the NP must compute N (generally $\max N = 256$) uniformly distributed pseudorandom values (let us call them $score_i$) using some packet fields ($vecv$ and the outgoing interface i. These values are later used as inputs for the mapping function that selects the forwarding port. Because we need to subtract $score_i$ from the weight according to (3-17), and because for the time being we assume that the weights are coded in 1-4-27 (sign-integer-fractional) fixed point notation, we have to use only the first 27 bits of the 32 produced by the uniform 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:

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

Let us finally recall some important properties of the function that we are looking for:

- The function must map (at least) a 32-bit input to a 32-bit output score. As input we will use some flow-dependent packet fields such as the source IP address or the source port number. In our case the destination IP address and the destination port number are constant, and consequently they do not increase the entropy of the hash scores.

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

- As we use flow-dependent inputs, the algorithm should generate the same score (*flow-order preservation*) for all the packets of the same flow. This avoids that packets of the same connection are being forwarded to different Real Servers. For UDP traffic this is not essential, but it is a fundamental issue for TCP connections.

- It should be possible to code the algorithm in picocode. The code will be executed N times for each packet because the NP needs a score for each port. Thus, the code requires a fast implementation.

- We cannot assume that the inputs are uniformly distributed: thus the output of the function should not show the effects of the bit correlation of the input.

What we propose is a list of hashing function that scramble and mix the input digits to obtain a new output score. We present several possible solutions to our problem, and try to evaluate the quality and the number of the operations involved in the hashing process.

3.2.6 Comparison criteria

First of all we have to select the comparison criteria. Let us suppose that we want to compare hash functions accepting 32-bit inputs. We cannot display a 2D graph showing the frequency F of the hash scores of all 2^{32} different keys because the range is too large, and we cannot provide a good resolution. Moreover that leads to problems understanding the correlation of the scores.

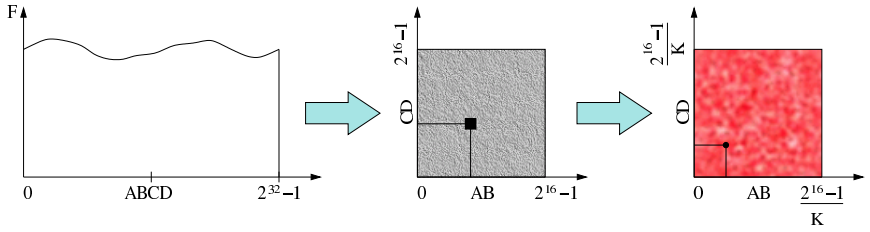

Figure 3.10: From 2D graphs to 3D ones

Instead, we propose a graphical approach that is useful for understanding the general behavior of the hash function. Because $2^{32} = 2^{16} \cdot 2^{16}$ we can transform the 2D graph into a 3D graph. In this case, the height of a point in the 2D graph $(0xAB, 0xCD)$ represents how many times the score $0xABCD$ was the result of the hash function. Unfortunately 2^{16} still is a large value, and it is difficult to display and handle all these data. Thus,

a compression is necessary. Let us denote this factor by K, say $K = 100$, and let us resize the squared graph to a $2^{16}/K \times 2^{16}/K$ squared one. In this case a $K \times K$ square of the previous graph is mapped onto only one point. If we want to maintain a 2D vision of the results, we can use colors to differentiate the frequencies. The color is computed using the HSB HSB notation. In particular the Hue is set to 0 (red) , the Brightness is set to 100, and the Saturation varies linearly from 0 (no frequency for this point) to 1 (max frequency value in this graph). This means that the greater a frequency a point has, the darker the red color that will be used for that point, and conversely the smaller the frequency of a point the lighter the red color that will be used. As we use a small set of the possible inputs, we use a grey background to differentiate the low-frequency points from those having no frequency.

We want to emphasize that this approach only helps us understand how data are spread and whether the correlation is small. A mathematical analysis using, for example, the χ^2 test could be done to verify the likelihood of the hash score distribution with an uniform one.

3.2.7 The input for the tests

First of all, we used a pool of 32 bit numbers as input (see Fig. 3.12) for all our tests. Using all possible combinations of four variables A, B, C, D that can vary in the range $[0, 255]$ with step 10, (i.e. A will be 0, 10, 20, ...) we can generate the input values using this function:

```
w32 getInputValue(w32 A, w32 B, w32 C, w32 D) {
   return A<<24 + B<<16 + C<<8 + D;
}
```

It is clear that a real trace file does not have a linear distribution of the IP addresses. With these simulations we only want to understand whether a specific hash function could be suitable for our needs. Thus, we check graphically whether the spreading is quite uniform and the collision number small, and in the end we will choose the best hash function.

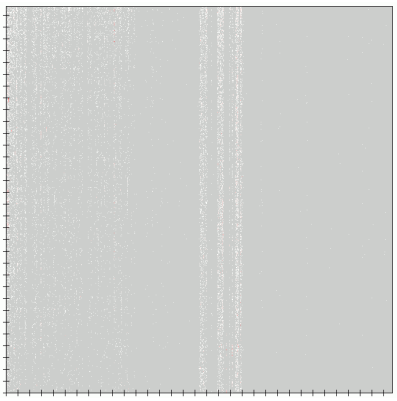

Figure 3.11: The distribution of the IP addresses of the NASA Kennedy Space Center WWW server

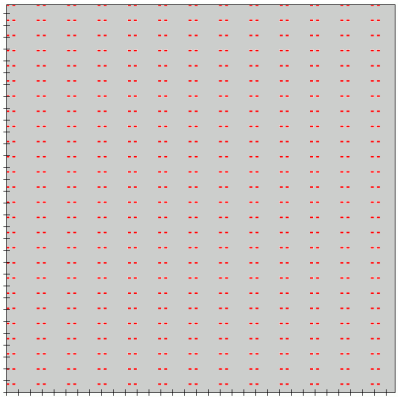

Figure 3.12: The uniform distribution of the IP addresses for the second input

During the simulation we saw that we had to use more than 32 bits. and thus we included the source port number in the hashing key. The distribution of the port numbers was generated according to the IANA assignments [21] for the clients port.

Finally, to simulate the behavior of the hashing algorithms, we used as input a specific real trace file [22] 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. In that log-file there are about 260000 different connections, which we parsed using a Java class. We have chosen this file even if it is quite old because the NASA server receives connections worldwide (we can have a consistent view of the IP address spread) and because for privacy reasons it is now impossible to obtain trace files with real IP addresses (they are provided as 16-bit hashed values). Fig. 3.11 shows the accumulation bands of the trace file: the more evident are generated by the IP addresses of the class 'C'.

3.2.8 Graphical analysis of the spreading

Previous hashing studies

Ref. [23] describes five possible direct hashing functions:

1. Hashing of Destination Address

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

2. Hashing using 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. Hashing using 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 five-tuple (source and destination IP address, source and destination port, and protocol ID):

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

5. The 16-bit CRC:

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

Ref. [23] shows that the first three hash functions work quite bad and are highly correlated with the address distribution. Moreover we use a unique IP address for the Server Farm (see Subsection 4.1.1), and thus the destination address is always the same. The fourth one is as good as the third but Ref. [23] defined the spreading-performance as not so interesting. The best performance in terms of un-correlation is provided by the last function. Unfortunately Ref. [23] used as input five parameters, but in our case the IP and port destination are always the same and thus the performance, as described in Ref. [23], could be not true. Moreover, there is no coprocessor for the CRC, and we should realize the function in picocode, which is too expensive in terms of CPU cycles.

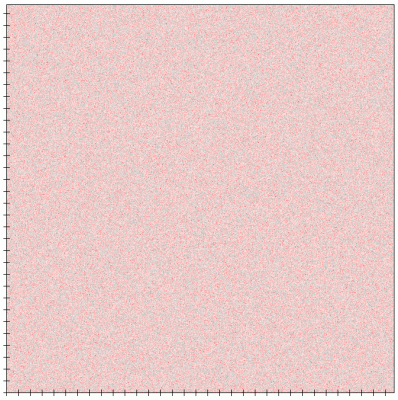

Figure 3.13: Jenkins' function with INPUT = IP

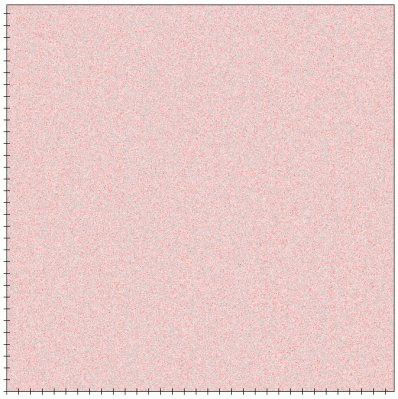

Figure 3.14: Wang's function with INPUT = IP

Knuth's standard reference

The standard reference for the integer hash functions is Ref. [24]. Knuth recommends the hash

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

hash = hash % table_size;
```

Unfortunately, that hash is only mediocre. The problem is the per-character mixing: it only rotates bits, it does not really mix them. It was demonstrated that every input bit affects only 1 bit of the hash. Moreover not efficient picocode implementation of this algorithm exists.

Jenkins and Wang's 32-bit Mix Function

Jenkins [25] provides some mix functions on his homepage that are freely available for commercial use. This is his famous 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); \
```

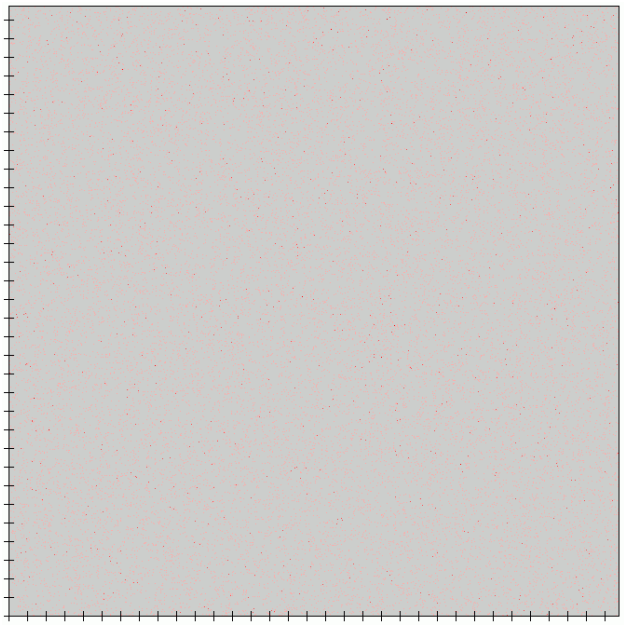

Figure 3.15: Jenkins' function with INPUT = IP (NASA input)

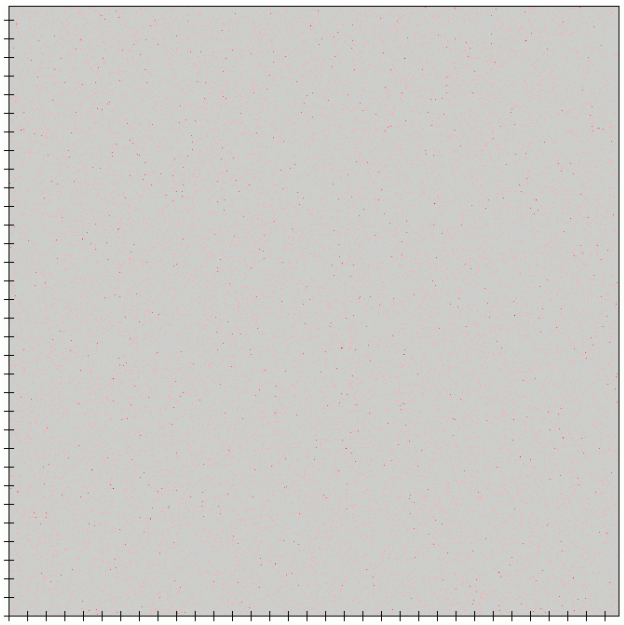

Figure 3.16: Wang's function with INPUT = IP (NASA input)

```
  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); \
}
```

where `c` initially contains the input key and at the end the hash result, and `a` and `b` are set by default to the 32-bit value of the golden ratio ($\phi = \frac{\sqrt{5}-1}{2}$). From this function Jenkins derived 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;
}
```

while Wang proposed another new (and faster) version in Ref. [26]:

```
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;
}
```

It can happen when executing `>>` that some compilers have an unexpected behavior such as inserting at the left '0' or '1' depending on the sign. To prevent this the author used the equivalent form `>>>` which forces the compiler to right shift the number and to put a '0' on the most significant bit.

As we can see in Figs. 3.13 and 3.14, the spreading is about uniform and for both the maximum frequency is 8. These functions have the best spreading, and the scores are highly independent, but they require too many cycles to be implemented in picocode. Also regarding to the NASA input, Figs. 3.15 and 3.16 show that the spreading is highly uniform. For both algorithm the greatest frequency is 3.

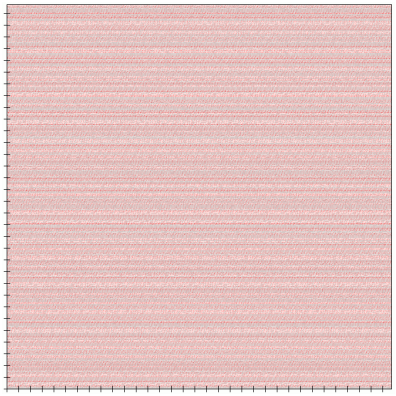

Figure 3.17: Knuth's function with INPUT = IP

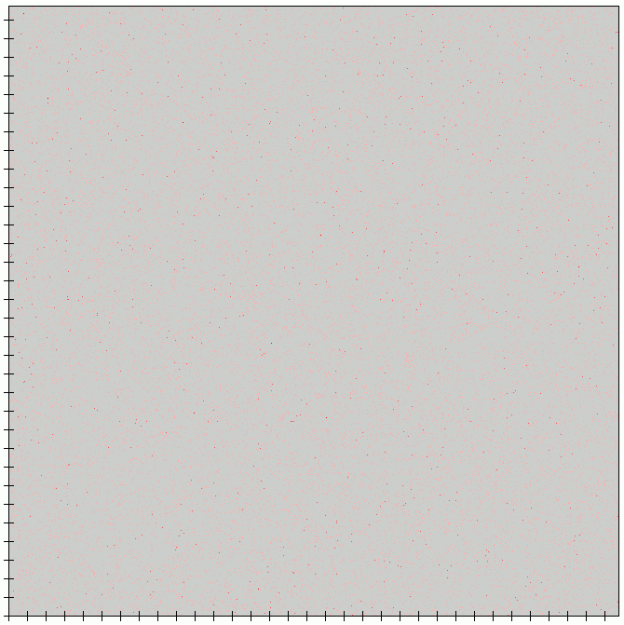

Figure 3.18: Knuth's function with INPUT = IP (NASA input)

Knuth's integer hash function

In Section 6.4 of [24], another multiplicative hashing scheme is introduced as a way to write the hash function. The key is multiplied by the golden ratio ϕ 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. Bad hash results are produced 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. In fact in the spreading graph (see Fig.3.17) we can see that only the lower 16 bits (on the y axis) are quite uniform. Moreover this method requires a multiplication by a fixed number. Even if we can reduce the number of operations with the Booth algorithm [27], the picocode implementation will require too many cycles.

The NP hash coprocessor (HC)

The NP also provides a customizable hash coprocessor as described in Section 2.1.2. The purpose of the coprocessor is to assist in the tree searches. This coprocessor can work asynchronously, and this is an important feature. While the coprocessor computes the hash score, the NP executes other instructions, improving device performance. Unfortunately we cannot provide more details about this coprocessor because it is confidential. Let us imagine it as a black box computing 32-bit hash scores from a max 192-bit input. We have analyzed the performance of the HC, and will present the graphical results.

Fig. 3.19 shows that the score-spreading is uniform and the max collision number is low (10) if we build the hash key in the form of "IP@ — IP@ + (Port_src $<$ 8)". In the figure the source port values were uniformly distributed according to IANA assignments [21].

Knowing that when a client must choose its port number, the smaller values are usually the most frequent ones (this is true for example for Linux Kernel) we thought that using uniformly distributed port numbers would not be a good assumption. Thus we tried to use an exponential distribution (Fig. 3.20) because in this way the smaller values are more frequent, but we have seen that the result is very similar.

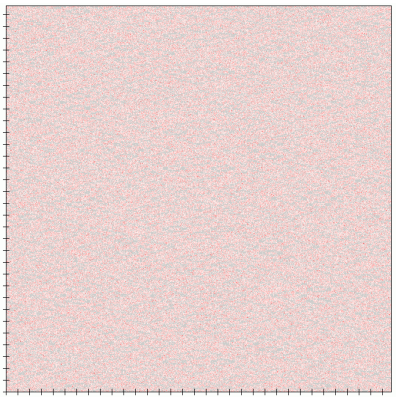

Figure 3.19: HC with INPUT = IP — IP + (Port_src < 8). Port value is uniform in [1024, 65535]

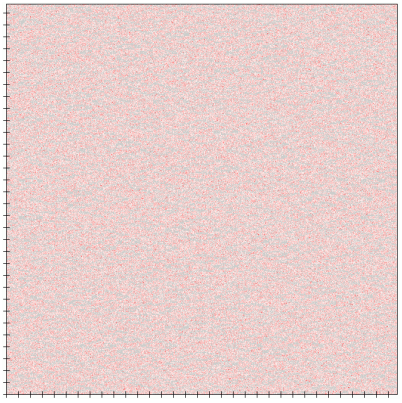

Figure 3.20: HC with INPUT = IP — IP + (Port_src < 8). Port values are distributed exponentially

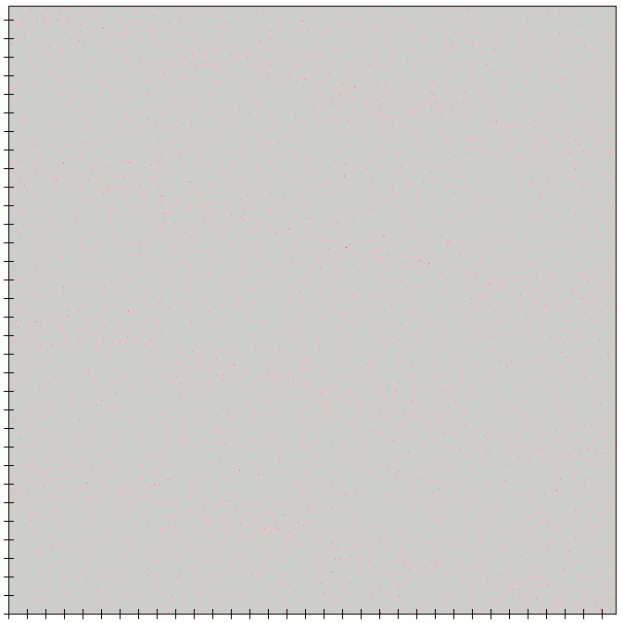

Figure 3.21: HC with INPUT = IP — IP + (Port_src < 8). (NASA input)

Finally we can conclude by saying that this solution is not only fast but also guarantees uniform and independent scores.

Jenkins's Multiple Mix Function

In an e-mail communication with Robert Jenkins, we obtained the following mix function:

```
#define mix(a,b,c,d,e,f,g,h) \
{ \
   a^=b<<11; d+=a; b+=c; \
   b^=c>>2;  e+=b; c+=d; \
   c^=d<<8;  f+=c; d+=e; \
   d^=e>>16; g+=d; e+=f; \
   e^=f<<10; h+=e; f+=g; \
   f^=g>>4;  a+=f; g+=h; \
   g^=h<<8;  b+=g; h+=a; \
   h^=a>>9;  c+=h; a+=b; \
}.
```

Calling it four times, we can obtain eight independent and uniformly distributed scores. However this function is clearly too expensive in terms of CPU cycles to be implemented in picocode.

Hybrid solutions

Hybrid solutions try to mix some of the algorithms presented to obtain efficient and faster versions.

We know that we need N uniformly distributed and independent scores. Let us suppose for now that $N = 32$. When a packet arrives, we can compute the score V using Jenkins' 32-bit Mix Function. We assume in this case that the V distribution is uniform. Then all 32 scores could be obtained using this approximation:

$$\begin{aligned} score_i &= (V + i \cdot \phi) \text{ RotateLeft } i \\ score_i &= (score_i + i \cdot \phi) \text{ RotateLeft } i \\ score_i &= (score_i + i \cdot \phi) \text{ RotateLeft } i \\ score_i &= (score_i + i \cdot \phi) \text{ RotateLeft } i \end{aligned} \tag{3-18}$$

Addition and rotation guarantee low values of the correlation among the scores. ϕ is the 32 golden-ratio value. The $i \cdot \phi$ offsets could be read from a static table. This technique could be applied to blocks of 32 outgoing links, and for each new block we can compute a new value V by adding, for example, a variable offset in the input key.

We tried to evaluate the cross-correlation among all variables in a small simulation. Let us suppose that we use 16-bit registers and want to generate 16 scores each time. Using (3-18) we computed the distributions of all $score_i$ when V assumes all values from 0 to $2^{16}-1$, and verified that the random variables $score_i$ are uniform. We finally computed the cross-correlation

$$\begin{aligned} Corr_{score_i,score_j} &= \frac{\sum_{i=0}^{2^{16-1}} (score_i - \overline{score_i}) \cdot (score_j - \overline{score_j})}{2^{16-1} \cdot \sigma_{score_i} \cdot \sigma_{score_j}} \\ &= \frac{\sum_{i=0}^{2^{16-1}} (score_i - 32767.5) \cdot (score_j - 32767.5)}{2^{16-1} \cdot \frac{1431655765}{4} \cdot \frac{1431655765}{4}} \end{aligned} \quad (3\text{-}19)$$

Table 3.1 shows some values of the cross-correlation matrix. We can see that this hybrid solution offers uniform numbers with small correlations. Simulations with the NP Java Simulator (we modified the Java class *mini_packet*, described in Subsection 3.3.3, in which the simulator computes the uniformly distributed hash scores) showed that it could be a good solution even if the correspondent picocode is not really fast (16 CPU-cycles with a rough approximation).

Table 3.1: Cross-correlation table

1.0000	0.0026	-0.004	0.3095	0.0037	0.0022	-0.092	-0.063	...
	1.0000	0.0085	0.0056	-0.176	0.0096	-0.003	0.0085	...
		1.0000	-0.000	0.0097	-0.124	0.0307	0.0014	...
			1.0000	-0.004	-0.006	0.2830	-0.029	...
				1.0000	-0.005	-0.005	0.0371	...
					1.0000	0.0041	-0.001	...
						1.0000	0.0016	...
							1.0000	...
								...

Another hybrid solution is using Jenkins's 32-Mix Function together with the hash coprocessor. For each arriving packet, the NP computes two 32-bit values (A and B) asynchronously using the HC, and synchronously a 32-bit value V with Jenkins' function. Then we obtain two 32-bit hash scores is

this way:

$$score_1 = UPPER(A) << 16 \mid UPPER(V)$$
$$score_2 = UPPER(B) << 16 \mid LOWER(V)$$

where $UPPER$ ($LOWER$) returns the highest (lowest) 16 bits. In this case we use only the first 16 bits of A and B, which, as shown, are the max entropy ones. Using the hash coprocessor we can work asynchronously, and the call to Jenkins's function produces two scores rather than one as it did before: the result is a fast code and a good distribution of the variables.

3.2.9 Conclusion

When comparing all the spreading figures with the 'standard' input, we see that the Jenkins and the HC functions offer the best results. Also with the NASA trace file, this performance is very similar: Figs. 3.15 and 3.21 have 3 and 4 as maximum frequency, and the spreading seems to be quite uniform. However, the HC was chosen as the feasible solution. The reason is the high speed (only few cycles are needed to load the input into the TSE, as shown in Subsection 2.1.2) and the asynchronous way which allows parallel operation and decreases the total number of cycles.

3.3 Dynamic load-balancing adaptation

In the first two sections we have described the Robust Hash Mapping and its drawbacks and finally we presented a PowerNP-tailored forwarding algorithm. In this section we will test our forwarding algorithm together with the control loop and verify its behavior under two particular traffic conditions: heavy traffic and wrong initial set of weights.

3.3.1 Motivation

Running only the forwarding algorithm does not guarantee that all outgoing links have the best utilization under all traffic conditions. One of the main reasons is that we made a strong approximation when we decided to

approximate the logarithm of a number with the first term of the Taylor logarithm series (see Subsection 3.2.4), and consequently we cannot obtain the desired load distribution on the outgoing links. We also know that the choice of the link depends on the incoming load (for example we use some packet fields, e.g. IP address or TCP/UDP ports, to generate the hash scores), and therefore particular traffic conditions could generate really unequal distribution of connection mappings. Moreover we also need a feedback in order to guarantee the system stability, to maximize the load over any outgoing links, and to prevent processor overload and packet dropping. Thus we need a mechanism that is able to check whether the system is going to be overloaded or underloaded, and consequently is able to modify some fundamental parameters so that we can guarantee high and constant system performance. This generally means that, after adaptation, the new connections will be forwarded to the underloaded links, which will be able to increase their utilization. In this way, the overloaded servers will have more time to empty their queues and return to a normal state.

3.3.2 The adaptation algorithm

Kencl and Le Boudec [15] divide this adaptation algorithm in two parts. The first one is the *triggering policy*. During this phase the Control Point gathers information about the link utilization from the NP. It analyzes the system-utilization trend and if it notes that there are links that are about to become overloaded or underloaded it triggers the adaption. The second phase is the *adaptation policy*, in which it modifies some link weights in such a way to maximize system utilization. For the full details of the algorithm, see Ref. [15].

Here we present a similar version that will be used in the project. We coded this version also for a Java simulation in order to verify in advance whether all our previous assumptions were correct.

Triggering policy

The Network Processor consists of N outgoing links to N Real Servers. By $\lambda_j(t)$, $\mu_j(t)$, and $q_j(t)$ we denote the number of packets arrived, the number of packets processed and the number of packets queued at the interface j in the time interval $(t - \Delta t, t)$. Thus we can define the *current link utilization*

$$\rho_j(t) \quad = \quad \frac{\lambda_j(t)}{\mu_j(t)} \tag{3-20}$$

(or the more complex version $\rho_j(t) = \frac{\lambda_j(t)+q_j(t)}{\mu_j(t)}$), and the *total system utilization*

$$\rho(t) = \frac{\sum_{j=1}^{N} \lambda_j(t)}{\sum_{j=1}^{N} \mu_j(t)} \tag{3-21}$$

(or the more complex version $\rho(t) = \frac{\sum_{j=1}^{N} (\lambda_j(t)+q_j(t))}{\sum_{j=1}^{N} \mu_j(t)}$). As in [15], in order to evaluate the status of individual processors we introduce a smoothed, low-pass-filtered processor utilization measure $\overline{\rho}_j(t)$ of the form

$$\overline{\rho}_j(t) = \frac{1}{r}\rho_j(t) + \frac{r-1}{r}\overline{\rho}_j(t - \Delta t), \tag{3-22}$$

where r is an integer constant (e.g. $r = 3$). A similar filtered measure is introduced for the total system utilization

$$\overline{\rho}(t) = \frac{1}{r}\rho(t) + \frac{r-1}{r}\overline{\rho}(t - \Delta t).$$

This filter is used to reduce the influence of short-term load fluctuations and to obtain information about the trend in processor utilization. Next let us introduce a dynamic *utilization threshold*

$$\epsilon_\rho^{'}(t) = \frac{1}{2}(1 + \overline{\rho}(t))$$

positioned midway between the current filtered total system utilization $\overline{\rho}(t)$ and the utilization of 1.

Ref. [15] also introduces a fixed hysteresis bound ϵ_h. We generally used $\epsilon_h = 0.01$. This prevents the adaptation when the individual link utilization stays within 1 percent of the total system utilization. Finally the *triggering*

threshold is introduced according to whether the system is over- or under-utilized:

$$\epsilon_\rho(t) = \max\left(\epsilon_\rho^{'}(t), (1+\epsilon_h)\ \overline{\rho}(t)\right), \quad \overline{\rho}(t) \leq 1$$
$$\epsilon_\rho(t) = \min\left(\epsilon_\rho^{'}(t), (1-\epsilon_h)\ \overline{\rho}(t)\right), \quad \overline{\rho}(t) > 1$$

Thus, using this threshold we trigger the adaption in the following cases:

$$\overline{\rho}(t) \leq 1 \quad \Rightarrow \quad \text{if } \left(\epsilon_\rho(t) < \max_j \overline{\rho}_j(t)\right) \text{ then adapt}$$
$$\overline{\rho}(t) > 1 \quad \Rightarrow \quad \text{if } \left(\epsilon_\rho(t) > \min_j \overline{\rho}_j(t)\right) \text{ then adapt}$$

Adaptation policy

Ref. [15] demonstrates that the weight adaptation also possesses the minimum disruption property. In fact, by changing the weights, the CP also changes the way how the forwarding algorithm distributes the new connections on the Real Servers. However, it could also happen that some existing flows are redirected to new Real Servers. Ref. [15] shows that the number of existing flows changing the destination Real Server after the adaptation is minimal. Ref. [15] uses the weights as described in (3-2), but we showed the strong relationship between (3-2) and (3-15) in Eq. (3-16).

Let us now analyze in which way the weights, such as (3-2), are adapted. If the system is going to be underloaded ($\overline{\rho}(t) \leq 1$), the CP computes

$$c(t) = \left(\frac{\epsilon_\rho(t)}{\min_j \left\{\overline{\rho}_j(t) \text{ — } \overline{\rho}_j(t) > \epsilon_\rho(t)\right\}}\right)^{1/N},$$

and the adaptation of the weights will be the following:

$$x_j(t) := \quad c(t)x_j(t-\Delta t), \quad \overline{\rho}_j(t) > \epsilon_\rho(t),$$
$$x_j(t) := \quad x_j(t-\Delta t), \quad \overline{\rho}_j(t) \leq \epsilon_\rho(t).$$

Otherwise if the system is going to be overloaded ($\overline{\rho}(t) > 1$), the CP computes

$$c(t) = \left(\frac{\epsilon_\rho(t)}{\max_j \left\{\overline{\rho}_j(t) \text{ — } \overline{\rho}_j(t) \leq \epsilon_\rho(t)\right\}}\right)^{1/N},$$

and the adaptation of the weights will be the following:

$$x_j(t) := \quad c(t)x_j(t-\Delta t), \quad \overline{\rho}_j(t) \leq \epsilon_\rho(t),$$
$$x_j(t) := \quad x_j(t-\Delta t), \quad \overline{\rho}_j(t) > \epsilon_\rho(t).$$

3.3.3 An approximate Java NP Simulator

Here we present some outputs of the forwarding and adaptation algorithms realized in Java. We created this code to verify whether the imprecision of the forwarding algorithm brings the system to some unstable situations. We will not explain all the details of this simulator. Right now it is sufficient to know the following:

- We suppose a NP with eight outgoing links (*OutQueue*), all having different rates. All input queues converge to a big FIFO queue (*InQueue*), and the NP extracts the packets only from this one. This queue is a rough approximation of a scheduler. All queues could be considered as infinite. The load percentages p_j are always [5%, 5%, 10%, 10%, 10%, 15%, 20%, 25%].

- An object *mini_packet* simulates a packet and contains 8 random numbers uniformly distributed within [0,1] (*mini_packet.number[]*). The *mini_packet* objects are queued in *InQueue*. Moreover by generating the *mini_packet.number[]* with a specific uniform number generator, we will be able to test whether this function suits our needs.

- The $UserP_i$ is a fixed value describing the expected percentage of packets that a user wants to forward on link i. We use an array of these values. *PktProc* is the maximum number of packets that the NP can forward on all links in 1 second. Thus, $Rate_i = PktProc \cdot UserP_i$ is the expected number of forwarded packets in one 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 CP starts the triggering policy looking at where the incoming packets were forwarded. The weights adaptation is instantaneous. NP and CP activity is synchronous: CP always checks after NP forwarding. The user sets the simulation time, which generally is more the 100 seconds.

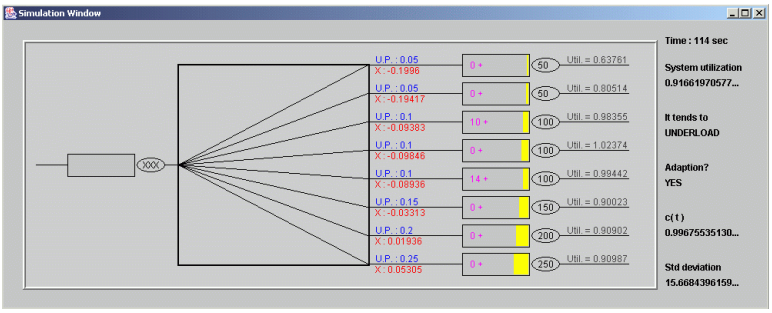

Figure 3.22: A screen shot of our Java NP Simulator

Every simulation second, the program displays the queue dimension, the number of packets forwarded to a specific queue, the utilization and the weights of that queue. At the end several graphs are displayed:

- Packet received standard deviation (per second):

$$Std(t) = \frac{\sum_{i=1}^{N=8} \{InQueue[i].forw(t) - InQueue[i].Rate(t)\}^2}{N} \quad (3\text{-}23)$$

- Per second dimension in packets of all queues
- Per second sum of the packets in all queues
- Per second utilization of all queues - see Eq. (3-20)
- Per second system utilization - see Eq. (3-21)
- Per second weights value.

Simulation results

Here we present the results of two simulations. We tested the Java NP Simulator near the critical threshold $\left(\rho(t) = \frac{\sum \lambda_i(t)}{\sum \mu_i(t)} \approx 0.95\right)$ to see what

happens when the system is congested. Then we observed the behavior of the system when the CP loads wrong 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 queue dimensions do not explode;
- the system and link utilizations are nearly unity;
- the standard deviation is in a range of acceptable values;
- the weights' values do not oscillate.

Congested system case

This 300-sec simulation shows a possible behavior of a congested NP. We set *PktProc* to 1000, and each second we generate

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

Finally $r = 3$ and $\epsilon_h = 0.01$.

This is clearly one of the possible situations of congestion. What we want to verify is what happens when the difference between the number of packets forwarded to link j and the rate of that link are about constant and nearly 0. In this case if the NP forwards the packets badly (e.g. too many packets to slower or congested links) we should see some queue congestion effect. This phenomenon does not happen because on average only 50 packets are queued, with peaks of 100 packets (Fig. 3.25). Moreover, system utilization (Fig. 3.26) does not oscillate and always remains in the range [-0.97, +1.03], and also the standard deviation (Fig. 3.27) is always in a range of acceptable values. Finally the weights (Fig. 3.24) attain values in the range $[initial\_value - 0.02, initial\_value + 0.02]$, and this confirms the validity of the proposed solution with the logarithm approximation.

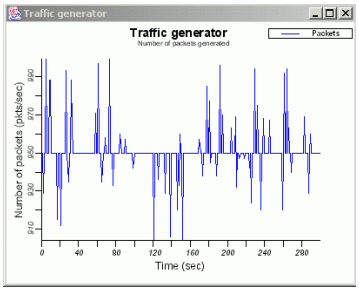

Figure 3.23: Number of packets at the ingress NP queue

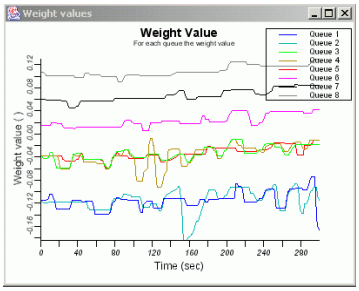

Figure 3.24: How the adaptation policy changes the weights' value

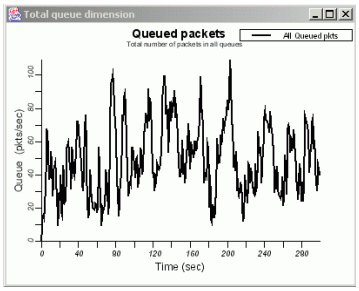

Figure 3.25: Total number of packets queued in the output queues

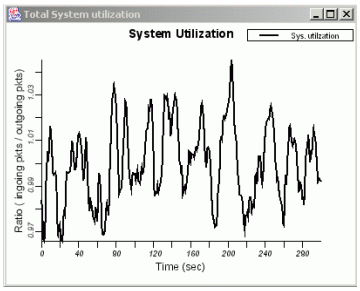

Figure 3.26: System utilization - see Eq. 3-21

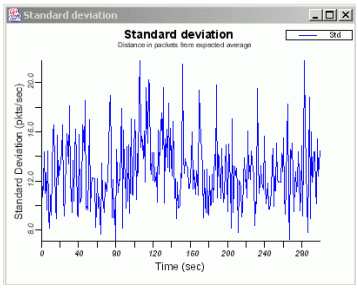

Figure 3.27: Standard deviation function - see Eq. 3-23

Wrong input weights case

This 300-sec simulation shows a possible behavior of an NP when the CP loads wrong initial weights. We set *PktProc* to 1000, and each second we generate

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

Finally $r = 3$ and $\epsilon_h = 0.01$.

We can easily see that wrong weights ($x_i = 0$) produce an initial wrong forwarding. In particular the first two outgoing links (the slowest ones) cannot process all the packets forwarded, which consequently are queued (Fig 3.29). But the algorithm reacts quickly by starting to change all weights. After the first 30 simulation seconds (this means that in the simulation CP adapts 30 times) their values tend asymptotically to the final ones. Moreover we can see that after $t = 30$ the first two link utilizations, Fig. 3.31, start to increase from 0 to 1, whereas the other link utilizations are almost equal to 1, and queue dimensions return to a normal state (note that we avoid the congestion case).

Finally note that the strong decrease of the standard deviation means that the control loop has a negative feedback, preventing queue dimension from exploding.

3.4 Handling TCP flows

Until now we described the forwarding and adaption algorithm assuming that we are working with UDP packets. Because the UDP protocol is connectionless we can forward the packets to different Real Servers without problems (considering no fragmentation). Conversely, with the 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 FYN or RST).

Let us give an example of a common problem. When the NP receives a packet for the server farm, it computes the outgoing interface (Target Port) by using the forwarding algorithm and the weights provided by the CP. If the set of the weights remains always the same, the forwarding algorithm assures *flow order preservation*, meaning that all packets of the same flow are forwarded to the same Real Server. In fact, only flow-dependent parameters, such as the source IP address and the source port, are used as input for the 'deterministic' (meaning same input, same output) hash function. But, say, at time $t = t_0$ the CP can change the set of weights (during the adaptation

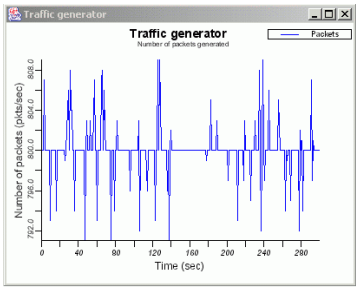

Figure 3.28: Number of packets at the ingress NP queue

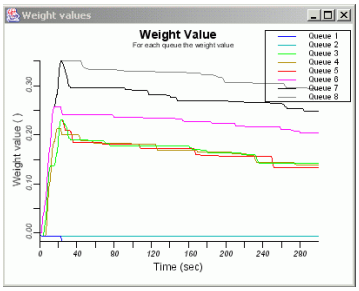

Figure 3.29: How the adaptation policy changes the weights' value

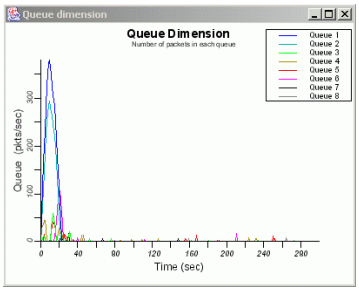

Figure 3.30: Packets queued in the output queues

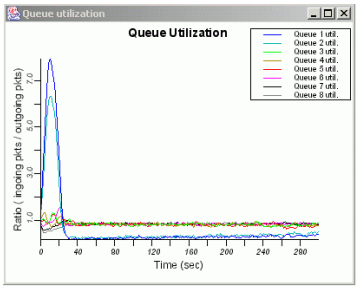

Figure 3.31: Queue utilizations - see Eq. 3-20

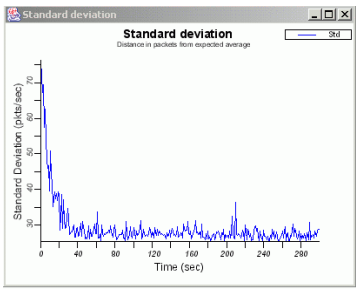

Figure 3.32: Standard deviation function

policy) and thus all the packets of that TCP connection arriving at the NP at time $t > t_0$ could be forwarded to a new Real Server, which can handle only new TCP connections but not an already existing one because it does not have any buffered state for it For example, in Fig. 3.33, the red flow is sent from the second to the third real server and, for this reason, it will be classified as a long-living flow.

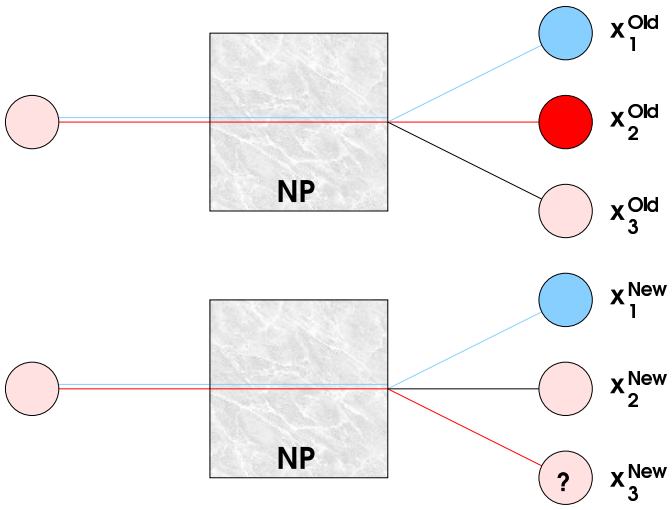

Figure 3.33: An example of long-living-flow

3.4.1 Long-living flows

Now we will try to describe the solution proposed by IBM-ZRL for handling such flows called *long-living flows*. Ref. [15] showed that after their adaptation the number of flows which are redirected to another Real Server is minimum (*minimal flow disruption property*), and thus we could drop all connections involved but this slows down the system performance, and the entire process is not transparent for the user. To solve this problem, the IBM-ZRL proposal states that we have to compute not only the Target Port (TP_{New}) using the set of new weights, but also the previous Target Port (TP_{Old}) obtained with the set of old weights. If the two final results are the same then the flow is not influenced by the change of weight set. If the Target Port is not unique, then the NP does not know where to forward the packet to, and we need to redirect the connection to the CP. Thus, because the forwarding algorithm cannot handle these flows, all these packets/connections must be handled and forwarded by the CP. The CP uses tables and timers to save and handle the state of these particular flows and decides where to forward them to. Each flow is handled until the user starts a new connection and the NP is able to compute unambiguous Target Ports.

According to IBM-ZRL proposal, the NP must therefore check first

whether the packet was already classified as belonging to the set of persistent flows. This can be done directly by the CP, which can configure the NP by installing, for example, some ingress classifier rules. A rule specifies, for example, the correct interface to which the packet matching the rule has to be forwarded. By adding or removing these rules, the CP can add or remove these particular flows. For all these packets, normal forwarding is disabled, and the NP uses the information provided by the CP for forwarding the packet.

Finally, the forwarding algorithm is carried out (using the sets of new and old weights) only for the non-filtered packets having as destination 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. Not-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. 3.34 shows the state machine proposed by IBM-ZRL for the NP side. Note that the NP performs only the *steady-state functions* and that we do all special computations in the CP (see Subsection 2.1.2). Let us now

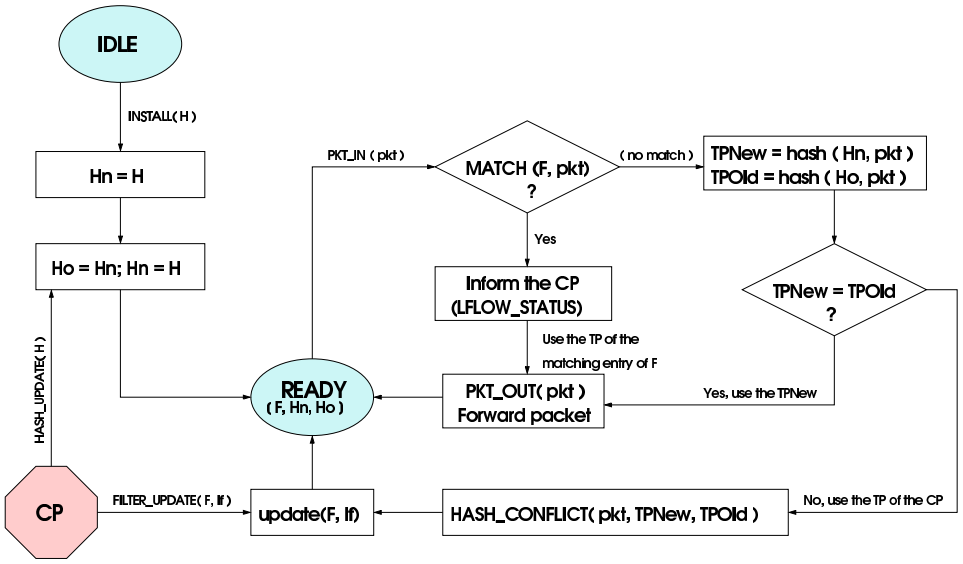

Figure 3.34: NP state machine

provide more details about Fig. 3.34:

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 the equivalent API which modifies the set of weights. This is done by overwriting the values of the old set (H_{old}) with them of the new set and installing in H_{new} the set passed as parameter (H);

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

MATCH is done directly by the hardware classifier which can filter and handle (LFLOW_STATUS function) the packets based on 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 real forwarding of the packet. The counter of the interface where the frame is forwarded must be also updated. This is necessary for collecting information for the adaptation policy.

Chapter 4

Implementation of the load-balancing algorithm

In this chapter we will focus our attention on the implementation of the proposed solutions on the PowerNP. We will present a typical simulation environment describing the Server Farm addresses, the interfaces, the ARP/IP routing tables, and the traffic generators and analyzers. Then, we will discuss the picocode solutions of the forwarding algorithm, and how to store, write and read the weights from the tables and how to solve the preemption problems in the weight update. Lastly we will talk about how to avoid correlation problems among the hash scores. There will be also small simulations testing whether the forwarding code works as expected. At the end we will propose a multi-level solution that really increases the speed of the algorithm (from linear to logarithmic complexity), and we will also propose a new and fast forwarding algorithm for the multi-level structure.

4.1 The simulation environment

In this section we provide details on the network settings. We did not work with real hardware. The network topology and the addressing have been emulated by the NPSim. As the *NETS* example presented in Subsection 2.4.2, the NPSim can also emulate multiple network and ethernet ports (i.e. the 'rethX' MAC-interfaces of Table 4.2). This emulation is completely transparent to the linux kernel which believes that all these cards are physical interfaces. Consequently, they appear in the ARP and IP routing tables.

4.1.1 The IP address assignment

The server farm must be not visible to Internet users. They must believe that they are always connected to the Virtual Server (that is the NP) even if it 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 private IP addresses, and the NP implements NAT functions. This solution decreases 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, and
   - change the IP source address of all outgoing packets to the Virtual Server IP address

   Note also that the NAT translation involves other operations such as re-computing packet checksum.

2. All real servers have the same IP address (that is the same as the virtual server). The real-to-virtual server connection must be considered as point-to-point. However this could for example provoke some problems in the ARP/RARP requests because the NP is linked to several nodes and shares the same IP address with them. To avoid problems we decided not to run ARP/RARP protocol on these interfaces but to put some fixed entries directly in the NP ARP tables for the IP/MAC mappings of the real server.

Solution 2 is certainly the better one in terms of speed and feasibility.

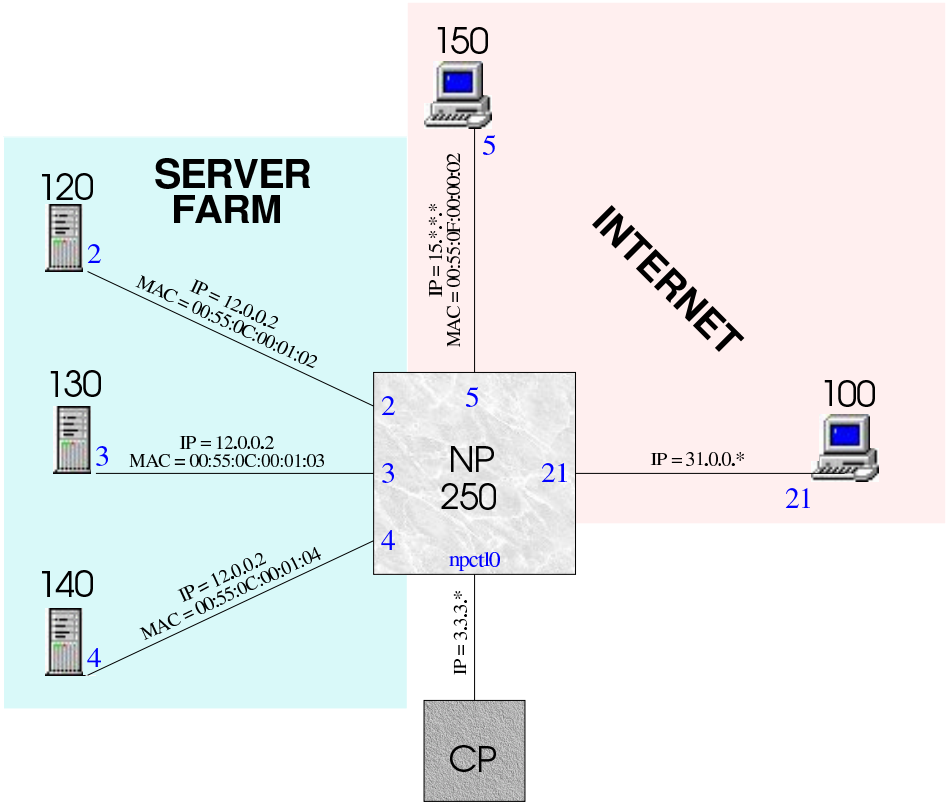

Figure 4.1: Example of a network topology

4.1.2 A possible network topology

Fig. 4.1 shows the network topology used during the tests. As we can see there are three real servers and two hosts simulating the internet users. Let us look at the IP routing table and the ARP table as shown in the Tables 4.1 and 4.2:

Table 4.1: 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

Table 4.2: 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

The most important linux commands for creating this topology are

```
arp -s 15.0.0.2 00:55:0F:00:00:02            # line 'O'

route add 12.0.0.2 reth2                     # line 'A'

arp -i reth2 -s 12.0.0.2 00:55:0c:00:01:02   # line 'L'
arp -i reth3 -s 12.0.0.2 00:55:0c:00:01:03   # line 'M'
arp -i reth4 -s 12.0.0.2 00:55:0c:00:01:04   # line 'N'
```

The lines 'B', 'C', 'D', 'E', 'F' and 'G' are set in a special file calls *ifcfg* (see appendix A). Note that line 'F' is the CP link. Line 'P' is the default gateway. Thanks to lines 'L', 'M' and 'N', 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 of the network card of that machine.

4.1.3 The Virtual Server IP address

In the software simulator, an application called 'reflection' uploads, during the booting phase, all the tables of the linux kernel directly into the NP kernel. For adding an entry into the NP tables, the CP uses some standard ASO functions. In particular for adding an IP routing entry, there is a specific API. This entry is composed by the IP address, the forwarding port (i.e reth3) and by a special 16-bit register of flags.

In the normal IP forwarding, the picocode uses the destination IP address of the received packet for a tree look-up and get the corresponding

forwarding interface and the register of flags. Depending on their meaning (i.e. 'BGP entry', 'redirection to CP', etc.), the picocode will execute the correct operations.

Looking at that register, we have seen that the four most significant bits are not used. We decided to use the 13th bit which has now the following meaning: "when the NP receives a packet and the tree look-up using the the destination IP address returns an entry having that bit set to '1' then the packet is sent to the server farm and the load balancing policy must be applied". Thus, if that specific flag is set, the picocode cannot use the forwarding port provided by the look-up tree, but it must run the forwarding algorithm to compute the TP.

It is easy to understand that the flag will be set only for the routing entry having the IP of the virtual server. Let us suppose that this address is '12.0.0.2'. We modified the API dor adding the routing entries, in a static way, in order to set that flag only when the IP routing entry (say `*req`) matches with the virtual server address.

```
np_uint32 VirtualServerAddress = 0x0C000002;
...
if (req->ipNetworkAddr.ip_addr == VirtualServerAddress)
    (&req->nextHop[0])->actionFlags |= NP_IPPS_ACTION_TO_RS
...
```

Line 'A' of Table 4.1 is used to insert a specific routing entry containing the virtual server address and to enable the load balancing policy for all packets having that destination address.

4.1.4 The weights representation

As we explained in Subsection 3.4, the CP provides two sets of weights to the NP: the new one x_k^n and the old one x_k^o. There are two important issues to define now: how to buffer the weights, and where to store them.

First of all note that, according to the final version of the forwarding algorithm presented in (3-17) we need to subtract the weight x_k from the

hash score $h(\vec{v}, k)$. x_k can be positive or negative, and, despite the hash score which always is in the interval $[0, 1]$, we have to provide some bits for the integer part of x_k. In section 3.2.2 we showed that using the weights computed using (3-6) we need at least there or four bits for the integer part (see Fig. 3.5). Let us assume four bits for the integer parts and one bit for the sign. However, considering that the sum of the weights (S) is a degree of freedom, if there are problems in representing some values, we can compute a set of weights which is equivalent and representable in our notation by only changing the value of S. In fact, we can easily see that if we translate all weights using a common offset, then the behavior of (3-17) will still be the same. For these reasons the only two possible fixed-point notations are as follows:

- 1-4-11: each weight is a 16-bit value. In this case loading the weights and handling them is easier, and the entire algorithm will require half the number of cycles. The information that will be buffered in a table can be compressed better; in fact we can read fixed-size blocks from tables, and if the weights have a small size we can read more weights with one access. The maximum decimal precision is $2^{-11} \sim 4.8 \times 10^{-4}$, which may not be not enough when the CP tries to adapt the weights.

- 1-4-27: each weight is a 32-bit value. In this case we will have some problems in loading, handling and storing the weights, but we will achieve a better decimal precision ($2^{-27} \sim 7 \times 10^{-9}$). This can avoid some representation problems, for example, when the CP tries to adjust the weights by a too fine quantity. Moreover, we could have metastable situations if the CP is not able to change the weights properly and consequently the weight will oscillate between two values. Finally, even if we use this notation we can buffer the data with the first notation (using only 11 bits for the fractional part), and check what the behavior of the NP would be. We will see also in the scalability paragraph that having more precision can be crucial in a multi-level hierarchy.

Thus, we will adopt the 1-4-27 solution for its versatility and precision.

The other problem to solve now is where to store these weights. The NP has, for each interface, a 32-bit scalar register available that we can use for storing the correspondent weights. But we would need at least 64 bits per interface because we have to store the old and the new value of the weight; moreover in a multi-level hierarchy we need some extra space for storing the tree structure. This solution is not feasible and it is clearly better to use a table. The NP can handle both static-size and dynamic-size tables. We do not require that the table changes its size, and moreover accessing a dynamic-size table is too slow because it requires first of all to access a table that stores its size. The static tables do not have specific interfaces for setting and writing the data (thus we need to code our own APIs), but clearly are the better solution. We can also choose the location in the memory of these tables: internal, SRAM or DRAM. The fastest memory is the internal, and we should use it because the table is read continuously. Naturally there is not much free space in that memory, and hence we can store only a small-size table. Looking at some of the possible configurations shown in Table 2.2 we can see that the internal RAM configuration with 512 bits per line guarantees that for each memory access the NP gets 8 new and 8 corresponding old weights.

4.1.5 Preemption on memory table accessing

There is a hidden problem occurring when CP changes the weight tables. Let us imagine that the NP reads a set of weights while the CP changes them. The result is that the forwarding algorithm is not applied correctly. One way to solve this problem is using a semaphore for locking the memory when the NP reads the table. However the reading operation is quite frequent and if the CP locks the table, this increases the connection latency because the packets stall in the NP. Considering that before parsing the memory lines containing weights the NP must know how many weights (N) must be parsed, we can imagine a different solution (see Fig. 4.2). First of all the

NP loads a line from the *MainTable* that contains a flag (*TableID*) and N. Using the flag, the NP loads the *Weight_Table*, which is the table with the valid weights (let us call the pointed table *Table_A*) and gets them from it. When the CP must change the weights, it performs the modifications on the other table (say *Table_B*) and then modifies the flag on the *MainTable*. Any thread already reading the old table will be not affected by this modification and only the new ones will read from the new valid table. Naturally this solution works only if the time between two table switchings (t_A in Fig. 4.2) is greater then the time needed by the NP to get the weights and forward the packet.

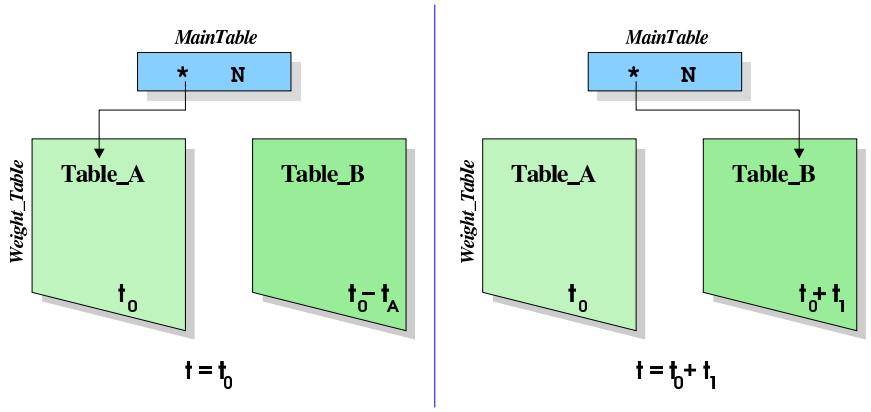

Figure 4.2: How to avoid preemption problems

4.2 The picocode implementation

4.2.1 The one level solution

Fig. 4.3 proposes the flow chart of the code implementing the forwarding algorithm presented in (3-17). Let us give more details on this coding. For example in the figure there are two small little dark green circles meaning that the picocode waits until a coprocessor, which has been called before asynchronously earlier, finishes computing the desired result. The flow-chart contains two cycles: one for reading and a 512-bit line from the memory and another one for running (3-17) for all the weights of that line. To implement these two cycles we used two indexes: M and N.

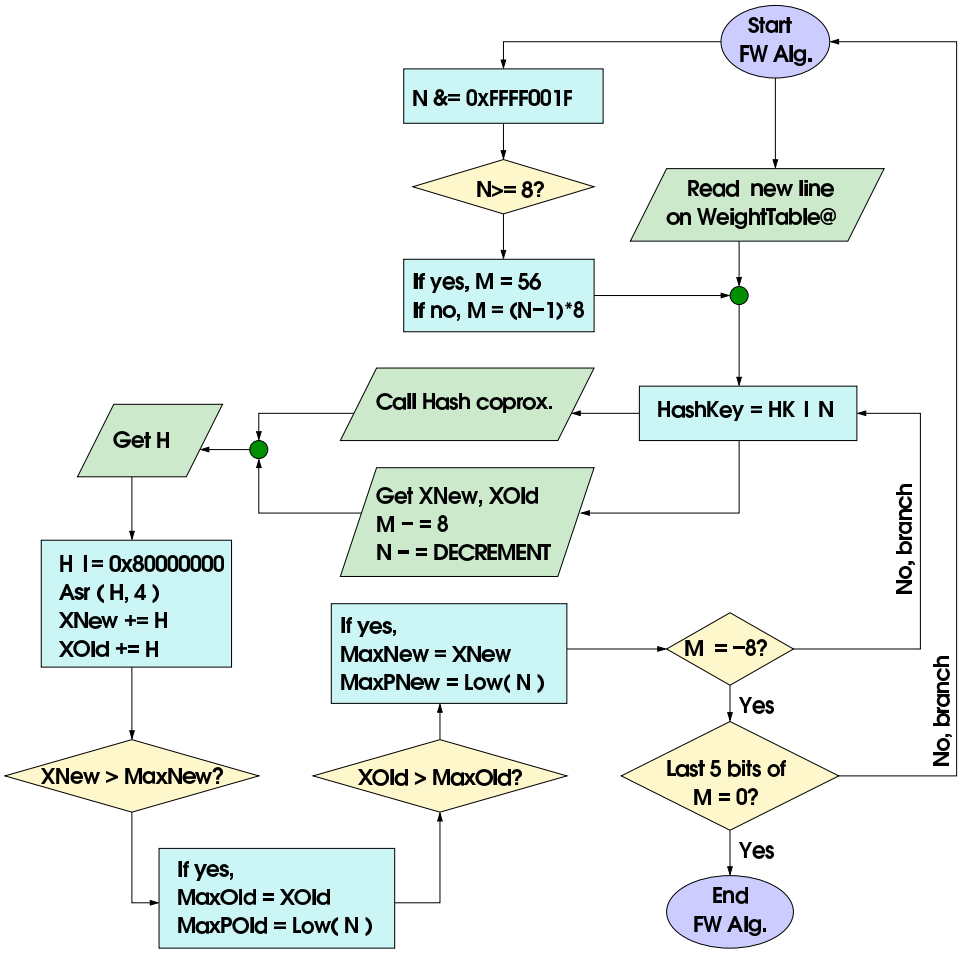

Figure 4.3: The core of the forwarding algorithm

Using M and decrementing it each time by 8, the NP can parse one by one all the pairs of weight (x_k^n, x_k^o) contained in one line of the *Weight_Table*. M is used as offset in the load operand which gets the 32-bit block from the memory . The reading starts on the right side of the line and finishes when M is negative. For this reason the starting value of M depends on the number of weights that the picocode expects to be in the line. For example (see case B of Fig. 4.9) if $N = 12$ then we need to parse two 512-bit lines: the first one contains certainly 8 pairs of weights (and thus $M = 56$), while the remaining 4 pairs (and thus $M = (N - 1) * 8 = 24$) are in the second one.

The initial value of N is the number of real servers of the server farm but then it is mainly used in the hash score generation. In recalling that the

Robust Hash Mapping requires that an ID (or an index or a tag, etc.) of the interface must be inserted in the hash key, we can use N in the hash key if the picocode decrements it by 1 each time it reads a pair of weights (x_k^n, x_k^o). In fact, in this way, N has a different value for each interface and it can be considered as an ID. Thus for generating the hash scores it is sufficient to append N at the end of the hash key, which will they be the following:

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

Unfortunately the hash scores must be uncorrelated, and if we use only five bits for representing N then the hash coprocessors provides highly correlated results. The easiest solution is to use a 32-bit register for storing N and decrementing it not by '1' but by a fixed 32-bit value, for example, the golden ratio. In this way we obtain N different indexes which, when plugged into the same hash key, give independent hash scores. Nevertheless, during the mapping algorithm, we also need to keep track of the winning interface, and an assignment such as $TP = N$ is not sufficient because the value of a specific TP is different from its ID. For example, the user can set that the fifth interface (TP = reth5) must be chosen if the $x_3 - h(vecv, 3)$ is the greatest value in the mapping algorithm. In this case, the third element is the winner, but the user has set the fifth interface for that element.

This means that we need another line (the *PointerBlock* line of Fig. 4.9) for the conversion from the winner port to the real TP of this port. For a fast indexing of this line, we need a value in the range [1,N] but now N is not in this range. We could use another variable for indexing this line, but there is another smarter and faster solution that we will explain using an example.

Let us consider a decrement value such as `0x9E377A01` (the same that we used in the picocode) and N equal to 6: Table 4.3 shows that the last eight bits are valid indexes for the Target Ports, while the first 24 are enough to obtain independent hash scores when N is appended in the hash key. Using this decrement value, when we need to save the winner port during the mapping algorithm, it is sufficient to save the least significant 8 bits (as

Table 4.3: Example of N values

interface	N
RS F	0x0000 00**06**
RS E	0x61C8 86**05**
RS D	0xC391 0C**04**
RS C	0x2559 92**03**
RS B	0x8722 17**02**
RS A	0xE8EA 9D**01**

shown in the blocks of Fig. 4.3).

Finally note that the hash coprocessor computes the hash score (H), which is a binary number in the interval [0,1). Because we need to subtract this number from the corresponding weights, it must be translated into the negative form. Instead of doing the 2'complement (costly), we left-insert five digits (sign + integer part) equal to '1'. The number that we create is exactly a negative number in 2'complement notation.

Simulation

We simulated a server farm with three host (IDs: 120, 130, 140) having reasonable load percentages equal to 20%, 30%, 40%. Using the program for sending packets we sent 6496 UDP packets using the packet information (i.e. IP addresses) collected by the NASA server [22]. As we can see, the

Table 4.4: Results of the one level simulation

ID	user percentage p_i	packets received	percentage
130	20%	1160	17.86%
140	30%	2052	31.59%
150	50%	3284	50.55%

results are close to desired, even if the number of real servers is not too big because of memory restriction of the linux workstation.

4.2.2 The multi-level solution

Unfortunately the complexity of the algorithm (3-17) and consequently of its picocode implementation is linear (we need to find the max among N

different values) and in spite of all code optimizations, the overall cycle performance is not excellent. We have already seen that there are two cycles: 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: for example as a rough estimation we can say that the inner one for parsing one weight requires 34 to 38 CPU cycles, which is quite a lot compared with the 1000 cycles that are the average for the whole normal IP forwarding. Moreover for users having a Server Farm, with for example, 50 web server, it is difficult to set 50 different percentages and for the NP is impossible to represent them because of the lack of precision.

We propose here a newer and smarter solution. First of all let us look at this example:

$$\max(A,B,C,D,E,F,G) = \max(\max(A,B,C,D),\max(E,F,G,H))$$
$$= \max(\max(\max(A,B),\max(C,D)),\max(\max(E,F),\max(G,H)))$$

This recursive approach changes the one-level structure (as shown in Fig. 4.4) into a multi-level hierarchy (as shown in Fig. 4.5).

The advantage is that using this new structure we need to compute $x_k - h(\vec{v},k)$ less often (for example five times instead of eight if we look at Figs. 4.4 and 4.5), and consequently we speed up the whole process. Nevertheless the increased speed is to the detriment of the ease of the new algorithm. First of all we need to use more than N weights. For example we can see that in Fig. 4.5 we have $8+4+2=14$ weights instead of 8. Using an example, we will show how to compute and manage them. Let us take the examples shown in Figs. 4.4 and 4.5. In the first case, users must set $p_A, p_B, p_C, p_D, p_E, p_F, p_G$, and p_H, which are the percentages of load that each real server should receive. In the second case, there are three levels, and the users must set the percentages of seven different *clusters* which are at the third level $(p_A, p_B), (p_C, p_D), (p_E, p_F)$ and (p_G, p_H), at the second level (p_{L11}, p_{L12}) and (p_{L21}, p_{L22}) and at the first level (p_{L1}, p_{L2}).

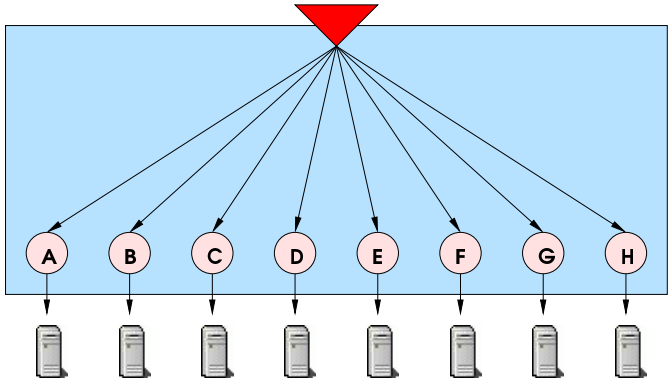

Figure 4.4: One-level architecture

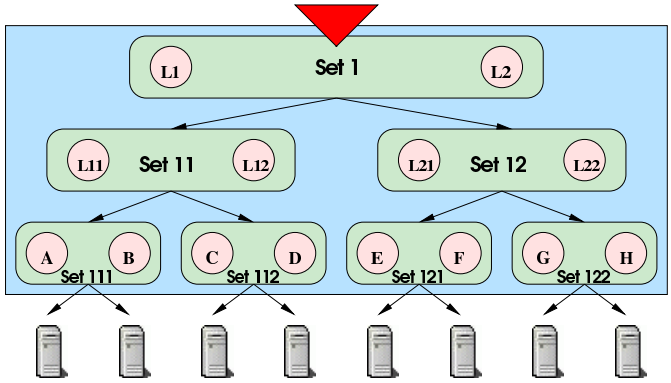

Figure 4.5: Multi-level architecture

The main difference is that now only the sum of the percentages inside a cluster must be 1. Moreover in the first case the real server A receives p_A of the incoming load, whereas in the second one it receives only the total percentage $p_A \cdot p_{L11} \cdot p_{L1}$. But in the second case we provide a better support for server heterogeneity to the user: for example the user can put the most powerful real servers into the cluster having the greatest total percentage and thus the three servers can manage their resources better. Nevertheless this solution presents also some drawbacks. As we have seen, we do not have one big cluster of eight elements but seven little clusters of two elements each. This means that the CP must run seven different adaptation policies. Let us imagine that, in the cluster $Set11$ of Fig. 4.5, the mapping algorithm forwards too many packets to the cluster $Set111$. What happens if the CP adapts the value of the weight x_{L11}? Certainly some flows will be redirected from the cluster $Set111$ to $Set112$. Unfortunately x_{L11} does not depend

directly on x_A and x_B and accordingly if the CP changes x_{L11} then some flows with real server A or B as endpoint will be redirected to C or D. This happens without x_A and x_B changing. Moreover, if the CP changes x_{L1} (because $Set1$ forwards too many flows to $Set11$), some flows directed to A, B, C and D are redirected to E, F, G or H. Thus, changing some weights of cluster 'X' will affect all the flows passing through that cluster.

The main problem is that we do not know which flows are redirected. Let us look at Fig. 4.6 and suppose that cluster A' is sending too many flows to cluster B. The CP adapts the weight 1' of cluster A', and let us assume that two flows must be redirected. If they are the red and the blue ones then everything works as expected, but if they are the yellow and the green ones then also the cluster B will be adapted. Thus, adapting at level n could cause a cascade of adaptations for all clusters below.

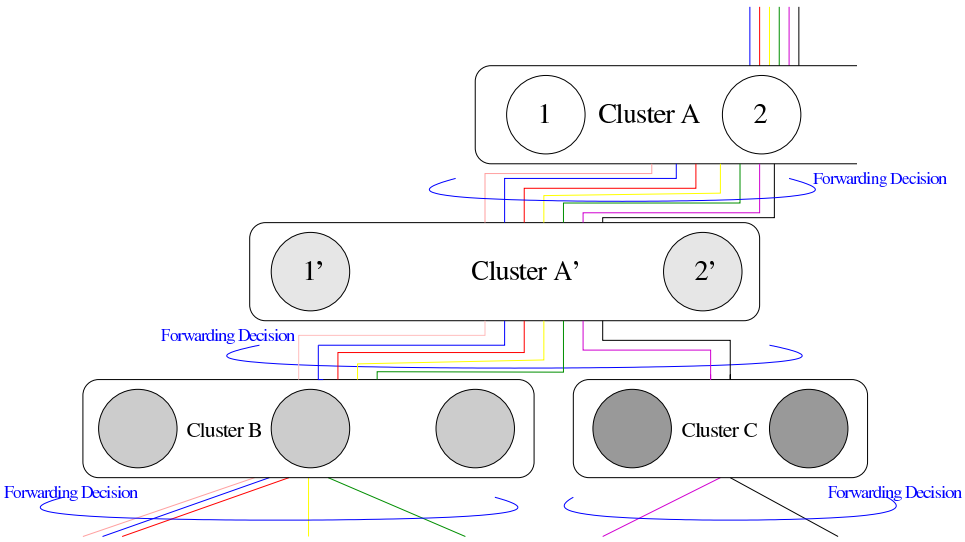

Figure 4.6: Example of flow distribution in the multi-level structure

The only solution is to consider the weights related to the internal clusters (i.e. nodes 1, 2, 1', 2' of Fig. 4.6) as quite constant. This can for example be achieved by increasing the hysteresis threshold in the adaptation policies related to the corresponding clusters (i.e. clusters A and A'). In this way the CP will start the adaptation on these nodes only if the forwarding performance is really bad (too much overload or underload). However when

they are adapted it is vital to avoid the adaptation on the lower levels for a certain period. In this way we avoid cascades of adaptations, and it is highly probable that, after this fixed period, with the arrival of new flows, the link utilizations come back to 'normal' values.

In view of we all these considerations, we believe it is better to make the choices listed below when aiming to improve the general performance of this structure:

- A reasonable depth of the tree. Too many levels in fact aggravate the redirection problem, resulting in bad performance. Choosing typically 2 (or at most 3) levels guarantees a fast execution and minimizes that problem.

- A reasonable number of branches. The main idea is to create a kind of autonomous sets with a sufficient number of real servers. They must not be too big because the users will have problems to set percentages and the NP will be not able to represent the weights due to the fixed-point notation, but we also do not want to create small set because this means increasing the depth of the tree too much. A reasonable maximum number of children of a node could be 16, but nothing prevent us from using a smaller number. For example with $N = 8$ the users can choose one level with 8 weights or 2 levels with 2 sets of 4 real servers.

The *Weight_Table*

With the multi-level solution we have to buffer not only the weights but also the tree structure. In order to minimize the number of cycles we modified the *Weight_Table* so that it contains all the information about the tree structure. Each cluster correspond to a block of the table. As shown in Fig. 4.9, a block contains in the first lines (called *Weight_lines*) the pairs of weights and in the last line (called *PointerBlock*) 16 32-bit cells. Because we decided to limit the number of branches (weights) to 16, there are at most 2 *Weight_lines*, each containing at most 8 weights. The picocode runs

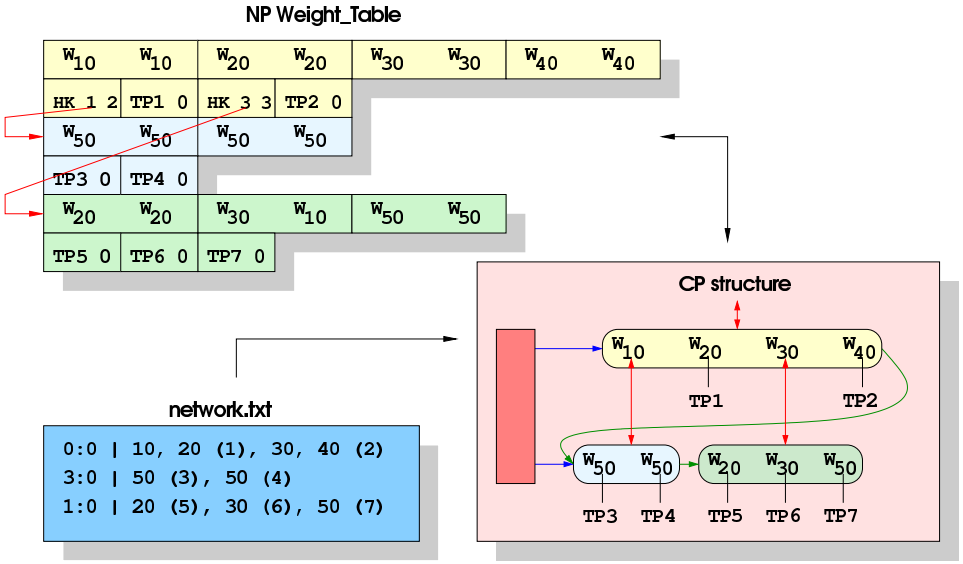

Figure 4.7: The user configuration, the CP structure and the NP table

algorithm (3-17) only inside the block (cluster) and produces two results TP^*_{New} and TP^*_{Old} that are used for indexing the PointerBlock line. Let us suppose that they are equal. The indexed cell ($PointerBlock[TP^*]$) informs the picocode depending on the position of corresponding node: if the node is not a leaf (last 5 bits of the cell are '0') the cell provides information on the number of real servers in next cluster (N), on its position in the table (the memory offset), and on what to add in the hash key in order to decrease the weight-correlation (HK); otherwise it gives the corresponding Target Port for reaching that particular real server. For example, looking at Fig. 4.8, we can see that in the yellow cluster the first and third node are not leaves (we must run another time the forwarding algorithm), while the second and the fourth are real servers.

We created an application that, based on user needs, builds this table. Figs. 4.7 and 4.8 show the links and the blocks used to create the tree structure inserted in a *.txt* file created by the user. As shown, each row of this file contains the position of the cluster in the tree structure, the description of the load percentages of each element of the cluster, and in brackets the corresponding TP if the element is a real server. The application checks the network topology inserted by the user, and computes, with linear complexity

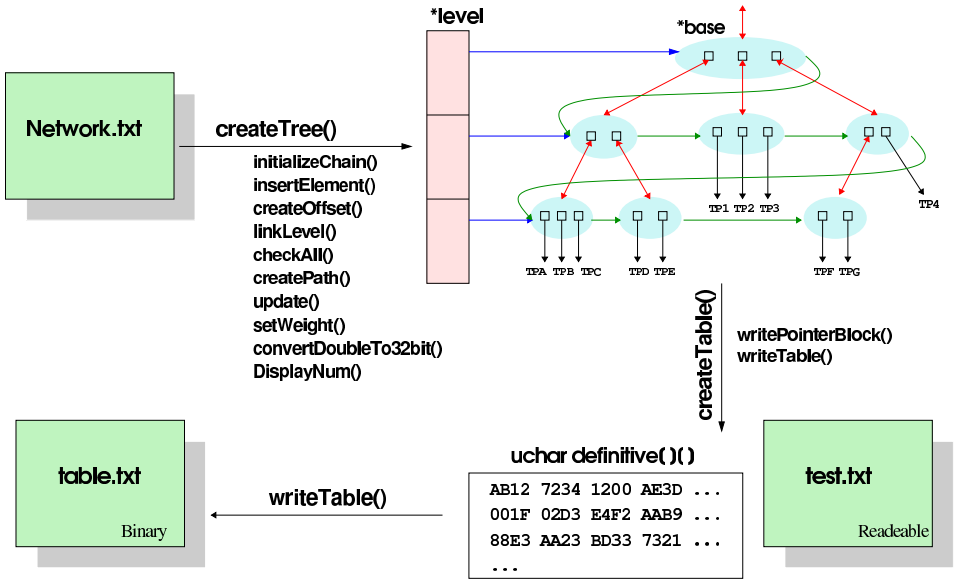

Figure 4.8: The CP application which creates the multi-level structure

all the weights and all the parameters of all *PointerBlock* lines (especially the memory offsets). The internal representation of the network topology is created for easy cluster handling, and it can be used in the CP, for example, for the adaption part.

Finally we want to emphasize the disposition of elements inside the *PointerBlock* line. We have already explained how the elements of *PointerBlock* are accessed by using N (see Subsection 4.2.1). Depending on its value (in particular whether it is greater or smaller than 8), they have a different position. For example, see Fig. 4.9, and look at the position of the red elements of case B.

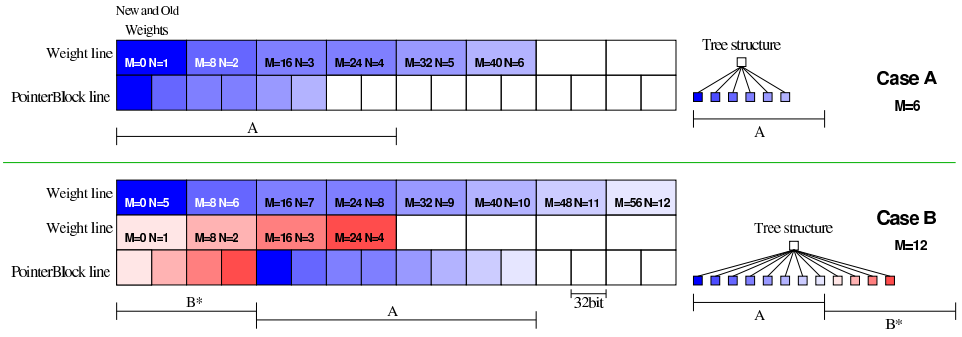

Figure 4.9: Example of a block of *Weight_Table*

Implementation details

This new multi-level structure is certainly more complex to implement, and poses several open issues that are still not solved. To obtain the final TP_{New} and TP_{Old}, the picocode must run the forwarding algorithm several times. Let us call the partial results, when the algorithm is applied to a cluster 'X', TP^*_{New} and TP^*_{Old}. Depending on their value there are three possibilities:

- They are the same, and the corresponding cell of the *PointerBlock* of cluster 'X' gives us the final TP.
- They are the same, and the corresponding cell of the *PointerBlock* of the cluster 'X' gives us the parameters to run the forwarding algorithm on a different cluster (that is a child of 'X').
- They are not the same, and we need to follow the two different paths until they both reach a leaf.

The first case is the easiest. As shown in Fig. 4.10, the picocode parses only one element of the *PointerBlock* table and, if necessary, restores all registers that are necessary for another computation. From a logic point of view, if TP^*_{New} is equal to TP^*_{Old} then the next cluster must be handled in the same way as the preceding one and the second case is the normal end of the first case.

The third case is instead quite complex: in the state-machine shown in Fig. 3.34 we can see that we must send not only the packet but also the new and old forwarding result (in this case of the whole process) to the CP. Thus, knowing that TP^*_{New} is not equal to TP^*_{Old} is not sufficient, we always have to compute TP_{New} and TP_{Old}. When we need to do this, the easiest way is to save all parameters concerning the old path and computing TP_{New}, then restoring them and computing TP_{Old}. Fig. 4.11 shows these two different areas. In the grey one, TP_{New} is computed, whereas in the yellow one the picocode, after restoring some registers, computes TP_{Old}. Let us finally note that the picocode also handles the special cases where TP_{New} and TP_{Old} are at a different depths.

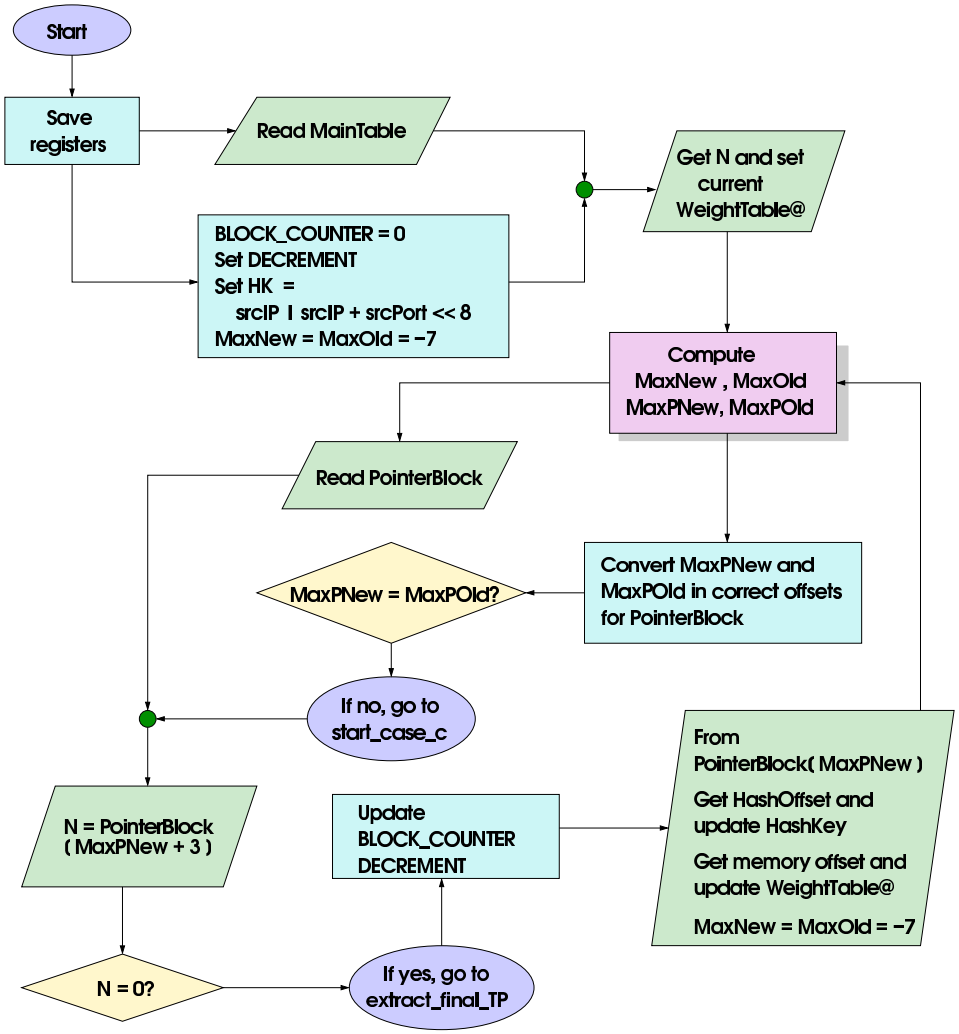

Figure 4.10: What happens at the end of a level when $TP_{New} = TP_{Old}$

Lastly we would like to discuss the meaning of updating the hash key when we change cluster. If the hash key for computing the scores inside the two clusters (a father and a child) is always the same, it can happen that the result of first cluster is highly correlated with that of the cluster-child.

This gives fixed results in the choice of the TPs and clearly bad performance. To solve this problem the simplest solution seemed to be to increase the key length depending the level/cluster position. Using this solution, however, it is really costly to compute where to append N in the hash key (see Subsection 4.2.1). The adopted solution is to add a 16-bit offset to the central part of the hash key. This seemed to be enough to un-correlate the

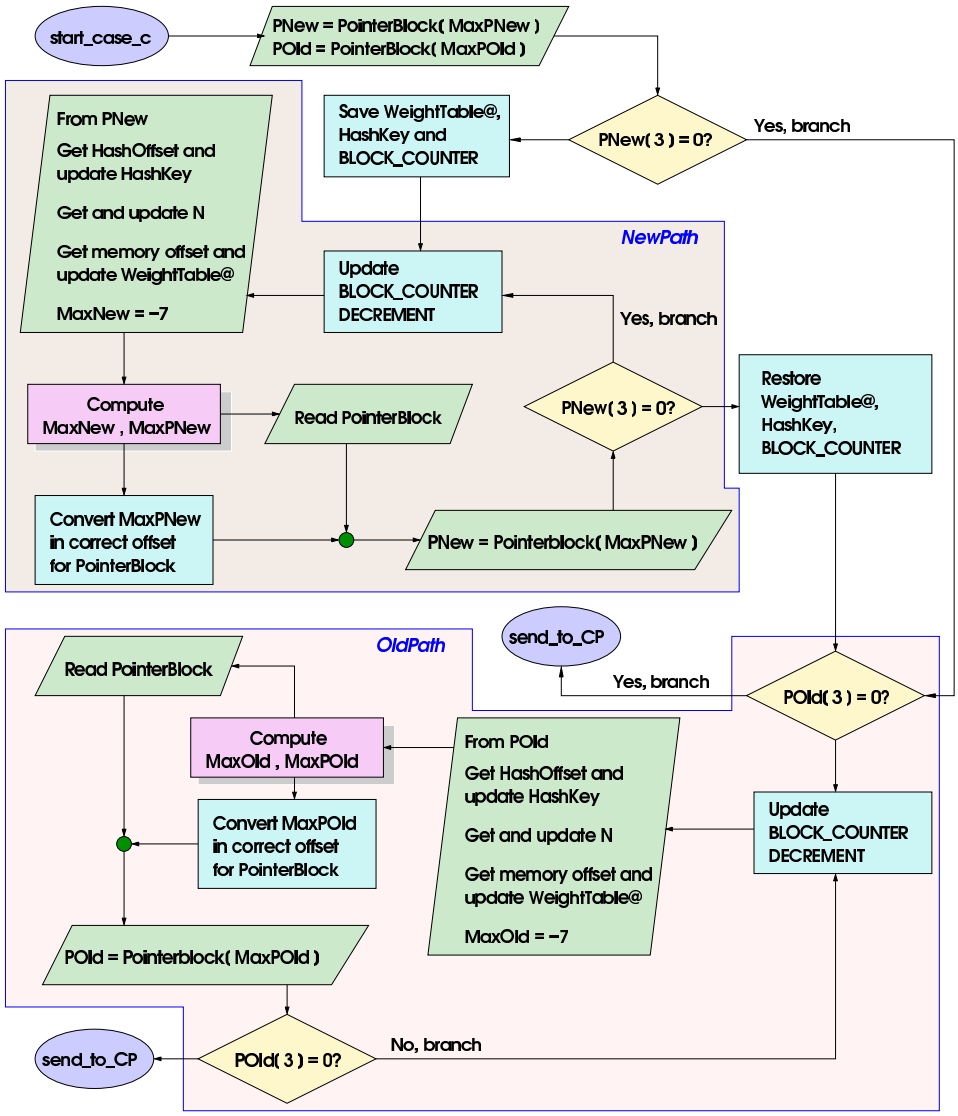

Figure 4.11: What happens when $TP_{New} \neq TP_{Old}$

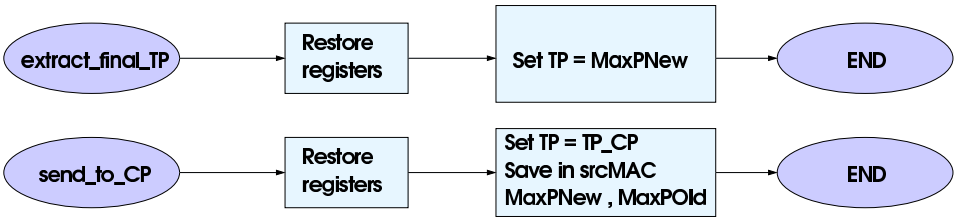

Figure 4.12: The packet is sent either to TP_{New} or TP_{CP} depending the FW algorithm result

results of different clusters. We did not do too much research for finding the best value of the addend. We used `0xBEAF` because there are many bits set

to 1, and this can better un-correlate the hash keys. Moreover depending on the level, it is possible to change, the value of the decrement of N and un-correlate the results better.

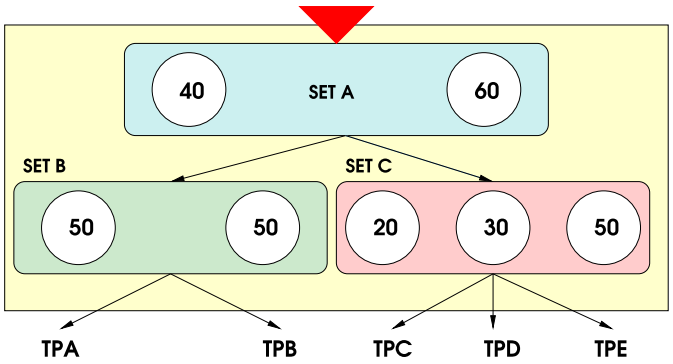

Figure 4.13: The topology for the multi-level simulations

Simulations

We simulated a topology as shown in Fig. 4.13, and once again sent UDP packets using the packet information (i.e. IP addresses) collected by the NASA server [22]. We present here the results of two simulations (see Table 4.5), and we show how the correlation between the results of different clusters can endanger the forwarding performance.

Table 4.5: Results of two multi-level simulations

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

In the first simulation we used `0x9E377A|01` (the 24-bit *golden_ratio*) as a decrement value for all clusters, and at the second level the first part of hash key was updated with $src\_IP@ \| src\_IP@ + src\_port << 8 + 0xBEAF << 8$. Even if the results are not as good as these obtained in the one-level simulation, the difference between desired and obtained percentages is not too big.

In the second simulation, N was decremented by `0x9E377A|01` at the first level, while at the second one the decrement was `0x61c886|01` (the 24-bit $1-golden\_ratio$), and the first part of hash key became $src\_IP@\|src\_IP@+ src\_port << 8 + 0xBEAF << 8$. With these parameters we have a high correlation between the scores of clusters A and C. In fact we can see that the TPC is never chosen. As we can see, reducing the correlation among clusters is an open problem, and only a practical test will suggest the best values. Moreover, considering that with this approach we reduce the number of weights parsed, the approximation of the logarithm using a table instead of the Taylor theorem, could be adopted.

4.3 Future work

We have already explained for the multi-level structure that a possible cascade of adaptations could happen when we adapt a weight. In this paragraph we propose another approach using a novel forwarding algorithm. First of all let us define two independent and exponentially distributed random variables $Z_1 \sim \exp(\lambda)$ and $Z_2 \sim \exp(\mu)$

$$f_{Z_1}(x) = \begin{cases} \lambda \cdot \exp(\lambda \cdot x) & \text{if } x < 0 \\ 0 & \text{if } x > 0 \end{cases} \quad \Bigg| \quad f_{Z_2}(x) = \begin{cases} \mu \cdot \exp(\mu \cdot x) & \text{if } x < 0 \\ 0 & \text{if } x > 0 \end{cases}$$

and let us compute the cumulative distribution of the maximum $Z \sim \exp(\sigma)$ between the two independent variables:

$$\begin{aligned} F_Z(x) &= P(Z \leq x) = P(max(Z_1, Z_2) \leq x) = \\ &= P(Z_1 \leq x, Z_2 \leq x) = \\ &= P(Z_1 \leq x) \cdot P(Z_2 \leq x) = \\ &= \exp(\lambda \cdot x) \cdot \exp(\mu \cdot x) = \\ &= \exp(x \cdot (\lambda + \mu)). \end{aligned} \tag{4-1}$$

As we can see also Z is exponentially distributed and $\sigma = \lambda + \mu$. This is an interesting property that we can use for creating a new mapping algorithm.

Let us consider the parameters of the exponential distributions as weights corresponding to the load percentages p_j set by the user. Let us recall that the inverse transform method says that

$$Z \sim \exp(\lambda) \sim \frac{\ln U[0,1]}{\lambda}$$

Thus we can propose a new mapping algorithm:

$$\begin{aligned} f(\vec{v}) &= j \\ &\Longleftrightarrow \\ \frac{\ln h(\vec{v},j)}{x_j} &= \max_{k \in [1,N]} \frac{\ln h(\vec{v},k)}{x_k} \end{aligned} \tag{4-2}$$

where the hash function $h(\vec{v},j)$ is uniformly distributed in $[0,1]$. Let us suppose that the sum of the weights x_j is equal to 1. Then we can demonstrate that the value of the weight x_j is equal to the corresponding load percentage p_j. In fact, referring to the same notation of the lemma demonstration in Subsection 3.2.1 and considering the exponential distribution of Z_n, we obtain:

$$\begin{aligned} q_n = P(Z_{(n)} \leq Z_n) &= \int_{-\infty}^{0} P(Z_{(n)} \leq x) \cdot x_n \cdot \exp(x \cdot x_n) dx \\ &= x_n \cdot \int_{-\infty}^{0} \prod_{i \neq n} P(Z_i \leq x) \cdot \exp(x \cdot x_n) dx \\ &= x_n \cdot \int_{-\infty}^{0} \frac{\prod_{i=1}^{N} P(Z_i \leq x)}{P(Z_n \leq x)} \cdot \exp(x \cdot x_n) dx \\ &= x_n \cdot \int_{-\infty}^{0} \frac{\prod_{i=1}^{N} P(Z_i \leq x)}{\exp(x \cdot x_n)} \cdot \exp(x \cdot x_n) dx \\ &= x_n \cdot \int_{-\infty}^{0} \prod_{i=1}^{N} P(Z_i \leq x) dx \\ &= x_n \cdot \int_{-\infty}^{0} \exp(x \cdot (x_1 + \ldots + x_N)) dx \\ &= x_n \cdot \int_{-\infty}^{0} \exp(x) dx \\ &= x_n \end{aligned} \tag{4-3}$$

The main property of this new mapping is shown in Fig. 4.14. The user sets the percentages (and thus the weights) $\lambda, \rho, \delta, \nu$ and μ on the lowest level. If $\lambda + \rho + \delta + \nu + \mu = 1$, then the corresponding weights, at the second

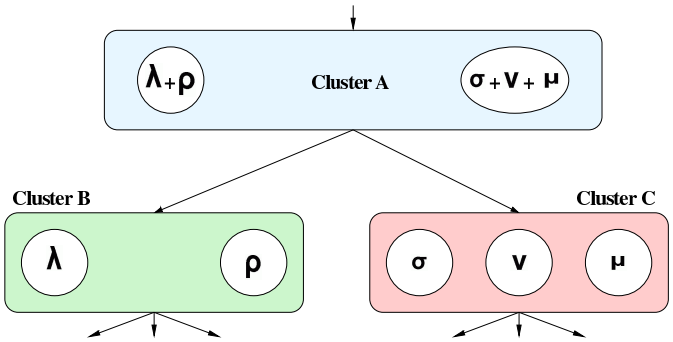

Figure 4.14: How to generate the upper level parameters

level, are $\lambda + \rho$ and $\delta + \nu + \mu$. Thus, if we need to adapt cluster A, it is sufficient to change the weights (percentages) in clusters B and C. In this way, we can decide which flows to redirect.

Open topics Unfortunately, the minimal disruption property on the adaptation of the weights is still an open topic, and the implementation of this mapping on the PowerNP seems to be difficult.

Chapter 5
Conclusions

In this thesis we presented a new load balancing algorithm that will be added as a feature of the IBM PowerNP. The proposed load balancing uses the Network Processor as a front-end device which transparently redirects the incoming TCP connections to a server farm based on some load percentages set by the user. The load balancer onsists of two main building blocks: the forwarding and the adaptation algorithm. The forwarding algorithm spreads the incoming flows according to the load percentages to the real server of the server farm. It uses fast hash functions to select the forwarding ports. The adaption policy verifies the link utilizations and, if necessary, changes some parameters (the weights) of the forwarding algorithm. In this way, because some connections change their end-point in the server farm after the adaption, the link utilization is always near unity.

In this thesis we have focused our attention on the forwarding block. A feasible solution has been proposed, described, and confirmed by initial tests.

In the first part of the thesis, we have analyzed the Robust Hash Mapping that is the main building block for creating high-performance forwarding algorithms. We have shown that, due to the weight-value distribution and the multiplicand operand, it cannot be implemented on the IBM PowerNP. We have found an equivalent formulation to compute new weights and a new hash score distribution. We have seen that the only way for generating this distribution is the inverse transform, which states that negative-exponentially distributed scores can be generated using the logarithm of uniformly distributed ones. The logarithm has been approximated with the

Taylor theorem, but we have proposed also another more precise method using tables. Finally we have graphically analyzed the uniformity of the spreading of some fast hash functions and finally selected the hash coprocessor functions because of its speed and versatility. This solution has been tested in some java simulations, and we have verified that the behavior was as expected. Finally we have presented the IBM proposal for handling the TCP flows.

In the second part of the thesis we have described some technical aspects of the picocode implementation. We have shown the network topology, the host addressing, the weights representation, and how we solved the preemption access problems. Then we have described the implementation of the solution and presented some initial results showing the correctness of our choices. However, we have noticed that, despite all code optimizations and the linearity of the algorithm, the proposed solution does not scale well. Thus, we have proposed a multi-level approach that is more scalable. We have described the more complex structures used to buffer the weights in the memory, and we have presented some testing results. With this multi-level approach, we have noted inter-cluster correlation problems. They can be solved by carefully setting some parameters added to the hash key and/or using the table approach for approximating the logarithm. The thesis closes with the discussion of some open topics, such as a new forwarding function that seems to fit the multi-level structure better.

Appendix A

The *ifcfg* file

```
#----------------------------
#       PTS interface configuration
PTSNAME=npctl0
PTSPORT=0x5555
PTSIPADDR=3.3.3.3
PTSIPMASK=255.255.255.0
MACADDR=03:B2:22:22:22:22
#----------------------------
#       NP Interface configurations
DEVICE=reth2
PORT=2
IPADDR=12.0.0.2
IPMASK=255.255.255.0

DEVICE=reth3
PORT=3
IPADDR=12.0.0.2
IPMASK=255.255.255.0

DEVICE=reth4
PORT=4
IPADDR=12.0.0.2
IPMASK=255.255.255.0

DEVICE=reth5
PORT=5
IPADDR=15.0.0.1
IPMASK=255.0.0.0

DEVICE=reth21
PORT=21
IPADDR=31.0.0.1
IPMASK=255.255.255.0
```

Bibliography

[1] D. Malagrino for Cisco Systems. Slides of secure and efficient e-business technologies. http://www.polito.it/∼dante/network-design/reti_di_campus/slide/09_Content.pdf, 2002.

[2] Jon Jenkins. How enhancing scalability with web farms. http://www.topxml.com/conference/wrox/2000_vegas/Powerpoints/jon_farms.pdf, 2001.

[3] Akamay web site. http://www.akamai.com/.

[4] Speedera web site. http://www.speedera.com/.

[5] Digital Island web site. http://www.akamai.com/.

[6] Madge.web web site. http://www.madgeweb.com/.

[7] Resonate white paper. Resonate Central Dispatch Application Note. http://www.resonate.com/solutions/literature/app_note_cd_plb.php, 2001.

[8] Windows & .Net Magazine. Windows NT Load Balancing Service. http://www.win2000mag.com/Articles/Print.cfm?ArticleID=15701, November 2000.

[9] Intel Corporation. F5 Labs BIG/ip Server Array Controllers Cookbook. http://developer.intel.com/design/servers/solsets/F5/f5_cb.pdf, 2001.

[10] Cisco documentation. Cisco Local Director. http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml, 2001.

[11] Radware documentation. Radware Web Server Director. http://www.peapod.co.uk/products/radware/wsd.html, 2001.

[12] IBM public documentation. NP4GS3 documentation. http://www-3.ibm.com/chips/products/wired/products/network_processors.html, 2002.

[13] K. Ross. Hash-routing for collections of shared web caches, 1997.

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

[15] Lukas Kencl and J. Y. Le Boudec. Adaptive load sharing for network processors. In *Proceedings of Infocom 2002*, New-York, June 2002.

[16] Joachim H. Ahrens and Ulrich Dieter. Efficient table-free sampling methods for the exponential, Cauchy, and normal distributions. *j-CACM*, 31(11):1330–1337, November 1988.

[17] K. G. Hamilton. Algorithm 780: Exponential pseudorandom distribution. *ACM Transactions on Mathematical Software*, 24(1):102–106, March 1998.

[18] George Marsaglia and Wai Wan Tsang. The Monty Python method for generating random variables. *ACM Transactions on Mathematical Software*, 24(3):341–350, sep 1998.

[19] J. F. Fernández and J. Rivero. Fast algorithms for random numbers with exponential and normal distributions. *Comput. Phys.*, 10:83–88, 1996.

[20] C. S. Wallace. Fast pseudorandom generators for normal and exponential variates. *ACM Transactions on Mathematical Software*, 22(1):119–127, March 1996.

[21] The Internet Assigned Numbers Authority (IANA). Port number assignments.
http://www.iana.org/assignments/port-numbers, April 2002.

[22] The Internet Traffic Archive. Trace file of nasa kennedy space center www server in florida.
http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html, 2002.

[23] Zhiruo Cao, Zheng Wang, and Ellen W. Zegura. Performance of hashing-based schemes for internet load balancing. In *INFOCOM (1)*, pages 332–341, 2000.

[24] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1998.

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

[26] Thomas Wang. Integer hash functions: principles and solutions.
http://www.concentric.net/∼Ttwang/tech/inthash.htm, October 2000.

[27] Safat G. Zaky V. Carl Hamacher, Zvonko G. Vranesic. *Introduzione all'architettura dei calcolatori*. McGraw Hill, Reading, MA, Ottobre 1997.

Acronym list

ASO Advanced Software Offering

CDN Content Delivery Network

CIA Common Instruction Address

CLP Core Language Processors

CP Control Point

DMU Data Mover Unit

E-EDS Egress En-queue/De-queue Scheduler

E-FCB Egress Frame Control Block

EDM Egress Data Memory

EPC Embedded Processor Complex

ePPC Embedded PowerPC Complex

FHE Frame Header Extension

FHF Frame Header Format

GPP General Purpose Processors

GPR General Purpose Register

HC Hardware Classifier

HSB Hue Saturation Brightness

I-EDS Ingress En-queue/De-queue Scheduler

I-FCB Ingress Frame Control Block

IDM Ingress Data Memory

IDS Ingress Data Store

NAT Network Address Translation

NP Network Processor

PPM Physical MAC Multiplexor

SWI Switch Interface

TB Target Blade

TP Target Port

TSE Tree Search Engine

TSR Tree Search Register

WBC Working Byte Count