UNIVERSITÀ DI PISA

Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in
Ingegneria dell'Informazione

Tesi di Dottorato di Ricerca

Fast Packet Processing on High Performance Architectures

Autore:

Gianni Antichi ________________

Relatori:

Prof. Stefano Giordano ________________

Prof. Franco Russo ________________

Anno 2011
SSD ING-INF/03

To my parents Renzo & Graziella & to my grandparents Ernesto, Piera, Lido & Fedora. Thanks for all you have done.

Acknowledgements

First of all, I want to say thanks to my friends. I shared with you many wonderful moments during these years. Thanks Lorenzo, Matteo, Alessio, Stefano, Sabina, Daniele, Alessandra, Laura, Frangé, Francesco, Paolo, Gioia and Alfredo. I also want to thank all the people in my research group. Thanks Christian, Gregorio, Andrea et al. I spent with you a lot of good moments and a lot of good meals. I would like to acknowledge Fabio and Domenico. Thanks guys. We were an incredible team. Hope to work another time with you in the future. I want to thank my tutors prof. Stefano Giordano and prof. Franco Russo. I am grateful to Dr.Andrew Moore for giving me the opportunity of interning in the Computer Lab at University of Cambridge where i met a lot a good people and learn a lot of things.

Sommario

La rapida crescita di Internet e la nascita sempre piú veloce di nuove applicazioni di rete hanno portato ad una difficoltá sempre maggiore nello sviluppo di reti IP ad alta velocitá con supporto anche per la qualitá del servizio (QoS). Per tale motivo la classificazione dei pacchetti e la intrusion detection hanno assunto un ruolo chiave nelle reti di comunicazione moderne al fine di fornire QoS e sicurezza. In questa tesi mostriamo alcune fra le piú avanzate soluzioni per lo svolgimento efficiente di queste operazioni. Cominciamo introducendo il NetFPGA e i Network Processors come piattaforme di riferimento sia per la progettazione e lo studio che per l'imple-mentazione degli algoritmi e, in generale, delle tecniche che sono descritte in questa tesi. L'aumento della capacitá dei link ha ridotto il tempo a disposizione dei dispositivi di rete per l'elaborazione dei pacchetti. Per questo motivo, mostriamo in questo lavoro differenti soluzioni che, o attraverso operazioni di "randomizzazione" ed euristiche o con la costruzione intelligente ed efficace di macchine a stati finiti, permettono ai nodi di rete di effetturare operazioni di IP lookup, classificazione e deep packet inspection in modo veloce su piattaforme ad alta velocitá come i Network Processor o il NetFPGA.

Abstract

The rapid growth of Internet and the fast emergence of new network applications have brought great challenges and complex issues in deploying high-speed and QoS guaranteed IP network. For this reason packet classification and network intrusion detection have assumed a key role in modern communication networks in order to provide Qos and security. In this thesis we describe a number of the most advanced solutions to these tasks. We introduce NetFPGA and Network Processors as reference platforms both for the design and the implementation of the solutions and algorithms described in this thesis. The rise in links capacity reduces the time available to network devices for packet processing. For this reason, we show different solutions which, either by heuristic and randomization or by smart construction of state machine, allow IP lookup, packet classification and deep packet inspection to be fast in real devices based on high speed platforms such as NetFPGA or Network Processors.

Contents

List of Figures

List of Tables

Introduction

The amount of Internet traffic and link bandwidth increase day by day, and this forces network devices to meet harder and harder requirements.
The growth of users and applications pushes researchers towards the development of novel and audacious ideas and modern and fast devices. The rise in links capacity reduces the time available to network devices for packet processing (for instance, on a gigabit link a packet has to be processed within about 0.7 μs). A simple solution is, of course, the adoption of massive parallelism. However, increasing the number of processing engines is an expensive approach and increases the memory bandwidth requirements. For these reasons, tasks like Packet Classification and Deep Packet Inspection are still a critical processing for network devices, thus permanently requiring improvements and new algorithmic solutions. Several solutions for these problems are described in this work; all of them take into account real applications and development in high performance platforms such as NetFPGA and Network Processors. The first chapter of the thesis introduces NetFPGA and Network Processors with greater attention to Intel ones which have been adopted in many of the works described herein. With chapter 2, we move into Deep Packet Inspection, by discussing solutions based on DFA and Bloom Filters. Then, chapter 3 introduces two novel perfect hashing schemes, which are useful in devices provided by a limited amount of memory. They are based on randomized techniques and show remarkable results. Chapter 4 describes IP-lookup and Packet Classification techniques based on heuristic, hash tables and DFA (Deterministic Finite Automata). Finally chapter 5 describes two ideas for low-cost IP Traffic Generation ad Monitoring at high-speed.

Chapter 1

Introduction to FPGA-based Networking Boards and Network Processors

In this chapter we introduce FPGA-based networking boards and Network Processors. The most popular of the former, NetFPGA [1] and Combo series [2], are presented with greater attention to the NetFPGA which have been adopted in many of the works described herein. Only a brief introduction of Combo6, for the Combo series is inserted. We introduce also a comparison among Network Processor Platforms with greater interest in Intel ones, used extensively in the works presented herein. NetFPGA and Intel Network Processor have always been taken as reference in the design and development of the proposed algorithms in this thesis.

1.1 FPGA-Based Networking Boards

The main idea of the FPGA-based cards like NetFPGA and Combo6 is to give developers a possibility to work with "open hardware" and use it in the same way as open-source software. The heart of these cards consists of one or more FPGA (Field Programmable Gate Array) chips, memories and other necessary components (power supply, IO chips, connectors etc.). Due to the flexibility of FPGA chips, the functionality of these cards can be easily (and quickly - within just several milliseconds) changed by loading a new design into the FPGA. This approach can be used for many different research and development projects.

Figure 1.1: The NetFPGA main core: a Xilinx Virtex II Pro FPGA.

1.1.1 What is an FPGA?

An FPGA (Field Programmable Gate Array) is a software programmable digital device. Field–Programmable means that the customer can configure it after manifacturing. FPGAs are usually a good trade-off between ASICs (Application Specific Integrated Circuit) and PAL (Programmable Array LOgic). FPGA offer the advantage of obtaining the same functions that an ASIC could perform while being amenable to modifications even after the chip is deployed in a product.
FPGAs are used in a great variety of applications like communications, automotive, consumer, etc. The designer can program them directly, achieving:

- a reduction of the project time
- a direct verify making use of simulations
- trails on the application field

A functionality error can be simply corrected by re–programming the device and the design environments are largely user–friendly. These are the reasons why FPGAs are usually preferred to ASICs in the project phase, while their high price and large power consumption are unconvenient for large–scale prodution. So this is the formula which is rapidly growing: FPGAs for experimentations and designing, and ASICs for productions.

1.1.2 FPGA Design and Programming

FPGAs are programmed with Hardware Description Languages (HDL) like Verilog or VHDL, or making use of a "schematic–entry" mode that can offer a fast and simplified approach with the same performances. Leader companies such as Xilinx and Altera give a complete development environment to support their entire product-line, so customer can just create their own application, simulate it and then download it to the FPGA platform to verify its functionality. Using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place–and–route, usually performed by the FPGA company proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA.

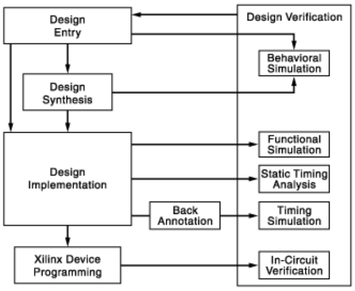


Figure 1.2: FPGA design flow.

The most common HDLs are VHDL and Verilog, although in an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are attempts to raise the abstraction level through the introduction of alternative languages. In a typical design flow, an FPGA application developer simulates the design in multiple stages throughout the design process. Initially the RTL description in VHDL or Verilog is simulated by creating test benches to simulate the system and observe results. Then, after the synthesis engine maps the design to a netlist, the netlist is translated to a gate level description where simulation is repeated to confirm the synthesis reports no errors. Finally the design is laid out in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist. As it can be seen, design verification, which includes both functional verification and timing verification, takes places at different stages during the design flow, giving the designer the chance to verify step by step his project without wasting time.

1.1.3 The Combo6

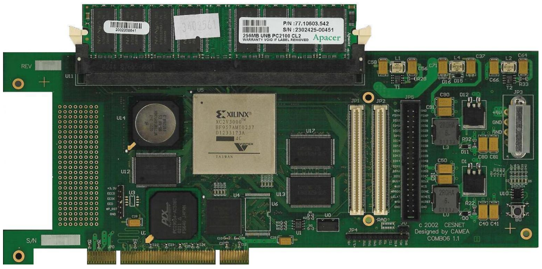


Figure 1.3: Combo6 upright view.

Combo6 developed in the Liberouter project [3] by CESNET in cooperation with the Faculty of Information Technology in Brno is a PCI card primarily dedicated for a dual-stack (IPv4 and IPv6) router hardware accelerator. It can be used in applications either alone (as a hardware coprocessor) or with add-on cards as an accelerated interface PCI card. It consists of Xilinx Virtex II FPGA, 2Mb TCAM, 256MB DRAM and 6Mb SSRAM. Various add-on cards can be used with Combo6 card. Add-on SFP card with 4 GE interfaces and 2 Virtex II FPGAs is an example of one of the Combo6 interface cards. The communication between interface and Combo6 card is realized via 100-bits wide 3-state bus with maximal clock frequency of 153MHz (maximal DDR transfer rate is approx. 3,8GB/s). Unfortunately the interconnection connector between Combo6 and its interface card is often source of reliability problems.

1.1.4 The NetFPGA

The NetFPGA platform has been developed in Stanford University by the High Performance Networking Group as an open platform available to developers worldwide. It is a reusable networking hardware with a simple modular design based on a low–cost FPGA platform. Its primary goal is to give students, teachers and researchers a fast and powerful tool to experiment new ways to process packets at line–rate. A set of reference designs is provided with the NetFPGA board, such as an IPv4 Router or a simple Network Interface Card. Several developers all over the world are using the platform to build advanced network flow processing systems.

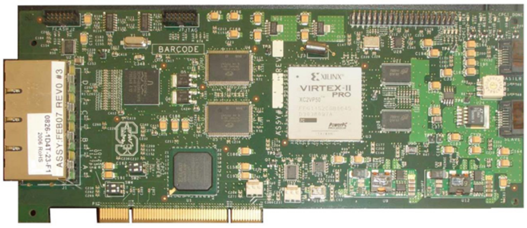


Figure 1.4: NetFPGA upright view.

1.1.4.1 Architecture

NetFPGA is a PCI card that plugs into a standard PC. As main components, the card contains an FPGA, four 1GigE ports and some buffer memory (SRAM and DRAM). A block diagram that presents the major components of NetFPGA platform is shown in figure 1.5. This platform contains one Xilinx Virtex2–Pro 50 FPGA which is programmed with user–defined logic and has a core clock that runs at $125Mhz$. The NetFPGA platform also contains one small Spartan II FPGA holding the control logic for the PCI interface to the host processor.
Two $18MBit$ external Cypress SRAMs are arranged in a configuration of $512k$ words of $36bits$ ($4.5Mbytes$ total) and operate synchronously with the FPGA logic at $125MHz$. One bank of external Micron DDR2 SDRAM is arranged in a configuration of $16M$ words of $32bits$ ($64MBytes$ total). Using both edges of a separate $200MHz$ clock, the memory has a bandwidth of $400MWords/second$ ($1,600MBytes/s = 12,800Mbits/s$).

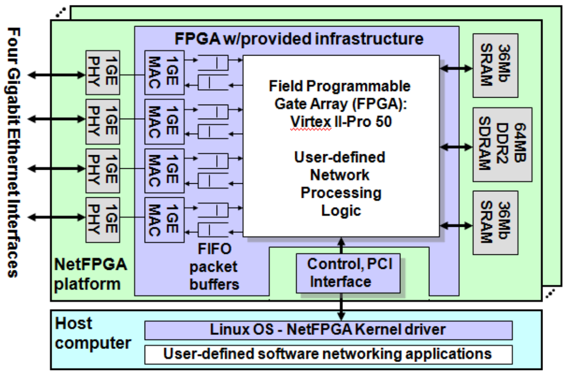

Four Gigabit Ethernet Interfaces
1GE PHY
1GE MAC
FPGA w/provided infrastructure
Field Programmable Gate Array (FPGA): Virtex II-Pro 50
User-defined Network Processing Logic
FIFO packet buffers
Control, PCI Interface
NetFPGA platform
36Mb SRAM
64MB DDR2 SDRAM
Host computer
Linux OS - NetFPGA Kernel driver
User-defined software networking applications

Figure 1.5: NetFPGA Block Diagram.

The Broadcom Gigabit/second external physical-layer transceiver (PHY) sends packets over standard category 5, 5e, or 6 twisted-pair cables. The quad PHY interfaces with four Gigabit Ethernet Media Access Controllers (MACs) instantiated as a soft core on the FPGA. The NetFPGA also includes two interfaces with Serial ATA (SATA) connectors that enable multiple NetFPGA boards in a system to exchange traffic directly without use of the PCI bus.

1.1.4.2 Reference Pipeline Details

Figure 1.6 shows the reference pipeline. The NetFPGA adopts the common approach of networking hardware which is generally arranged as a pipeline through which packets flow and are processed at each stage. Stages are interconnected using two point–to–point buses: the packet bus and the register bus.
The packet bus transfers packets from one stage to the next using a synchronous FIFO packet-based push interface, over a 64–bit wide bus running at 125Mhz (an aggregate rate of 8Gps). In this scenario, stage i pushes data forward when the next stage has space for a packet word (i.e.: the FIFO is not full). The FIFO interface has the advantage of hiding all the internals of the module behind a few signals and allows modules to be concatenated in any order. It is the simplest interface can be used to transfer information and provide flow control while still being sufficiently efficient to run designs at full line–rate.
The first stage in the pipeline consists of several queues (Rx queues). These queues receive packets from IO ports such as the Ethernet ports and the PCI over DMA and provide a unified interface to the rest of the system. These ports are connected into a wrapper called the User Data Path which contains the processing stages. The current design has 4 Ethernet Rx queues and 4 CPU DMA queues. Packets that arrive into CPU DMA Rx Queue X are packets that have been sent by the software out of interface nf2cX. In the User Data Path, the first module a packet passes through is the Input Arbiter. The input arbiter decides which Rx queue to service next, and pulls the packet from that Rx queue and hands it to the next module in the pipeline: The output port lookup module. The output port lookup module is responsible for

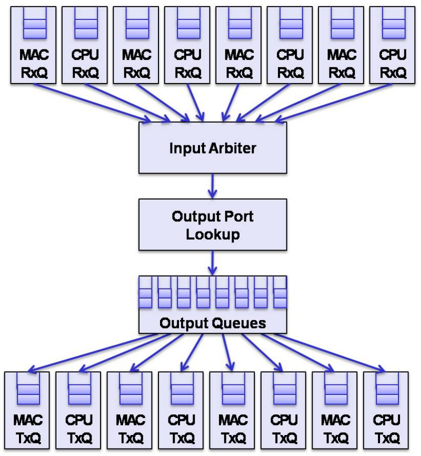


Figure 1.6: The Reference Pipeline.

deciding which port a packet goes out of. After that decision is made, the packet is then handed to the output queues module which stores the packet in the output queues corresponding to the output port until the Tx queue is ready to accept the packet for transmission. The Tx queues are analogous to the Rx queues and they send packets out of the IO ports instead of receiving. Packets that are handed to DMA Tx queue X pop out of interface nf2cX.

1.1.4.3 Life of the Packet

Packets enters and exit the pipeline through various Receive and Transmit Queue modules respectively. To keep things simple, the interface is packet–based. Modules are not required to process multiple packets at a time, and they are not required to split the packet header from its data (altough a module is free to chose to do so internally). As the packet moves from one stage to the next, a stage can modify the packet itself and/or parse the packet to obtain information that is needed by a later stage for additional processing on the packet. This extracted information is prepended to the beginning of the packet as a 64–bit word which is called *module header* and is uniquely identified by its ctrl word from other module headers. Subsequent stages in the pipeline can identify this module header from its ctrl word and use the header to perform additional processing on the packet. The ctrl world is non-zero for module headers and distinguishes module headers from each other when there are multiple module headers. When the actual packet received starts after the module headers, the ctrl word is reset to 0, and then at the last word of the packet, the ctrl lines will indicate which byte is the last byte in the last word. The Rx Queues create a module header when they receive a packet and prepend it to the beginning of the packet. The Rx queues store the length of the packet in bytes at the lowest $16bits$ of the module header, the source port and the packet length. The Input Arbiter selects an Rx queue to service and pushes a packet into the Output Port Lookup module. The Output Port Lookup module decides which output port(s) a packet goes out of and writes the output ports selection into the module header. The Output Queues module

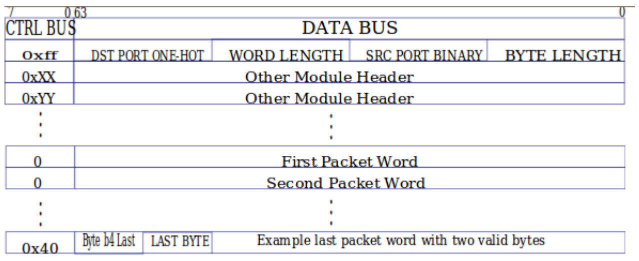


Figure 1.7: Format of the packet passing on the packet bus.

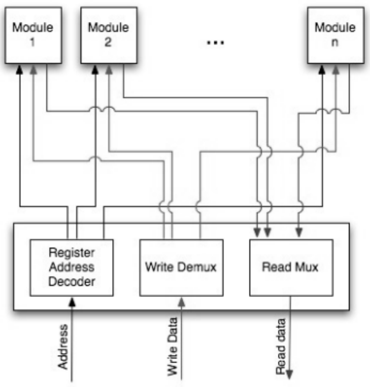


Figure 1.8: Register access system.

looks at the module header to decide which output queue to store the packet in and uses the lengths from the module header to store the packet efficiently. After the packet is removed from its output queue and pushed into its destination Tx Queue, the module header is finally removed before sending the packet out of the appropriate port. Packets on the packet bus are formatted as shown in figure 1.7.

1.1.4.4 Register Pipeline

The register bus provides another channel of communication that does not consume Packet Bus bandwidth. It allows information to travel in both directions through the pipeline, but has a much lower bandwidth. A common register interface, done by memory–mapping, make the hardware registers, counters and tables visible and controllable by software. The memory–mapped registers, in turn, appear as I/O registers to the user software that can access them using ioctl calls.
The register bus strings togheter register modules in each stage in a pipeline daisy–chain that is looped back in a ring. One module in the chain initiates and responds to requests that arrive as PCI register requests on behalf of the software. However, any stage of the chain is allowed to issue register access requests, allowing information to trickle backwards in the pipeline, and allowing stage i to get information from stage $i + k$. This daisy–chain architecture is preferable to a centralized arbiter ap-

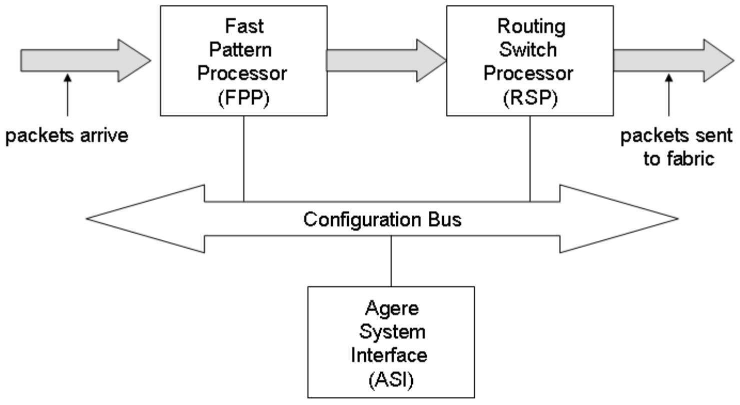

Fast Pattern Processor (FPP)
Routing Switch Processor (RSP)
packets arrive
packets sent to fabric
Configuration Bus
Agere System Interface (ASI)

Figure 1.9: Architecture of NP Agere.

proach because it facilitates the interconnection of stages while limiting inter–stage dependencies. Figure 1.8 shows this register bus architecture.

1.2 Comparison among Network Processor Platforms

Nowadays no official standard for network processors yet exists. Each vendor has proposed its specific solution. The common features of all proposals are a hierarchy of processors, a series of memory devices with different sizes and latencies, a low-level programmability. The target is a platform for networking applications with low time to market and high time in market, thanks to a high flexibility and a easy programmability. In this section, we first present a comparison among the available network processor platforms and then a detailed description of the hardware reference of our activity, the Intel IXP2XXX family.

1.2.1 Multi-chip Pipeline (Agere)

Agere System Incorporated (i.e. the microelectronics branch of Lucent Technologies) presents a NP family called Payload Plus [4]. It has three interesting features: a multichip architecture, a programmable classifier, a flexible management of input data.

1.2.1.1 Architecture

The Agere system is composed by three different units. Fig. 1.9 shows the interconnections among the different chips and the data flow through the consequent pipeline. The Fast Pattern Processor (FPP) and the Routing Switch Processor (RSP) establish the basic pipeline for fast data path processing. The ingress packets are forwarded to the FPP, which sends them, along with an instruction set, to the RSP. The packets

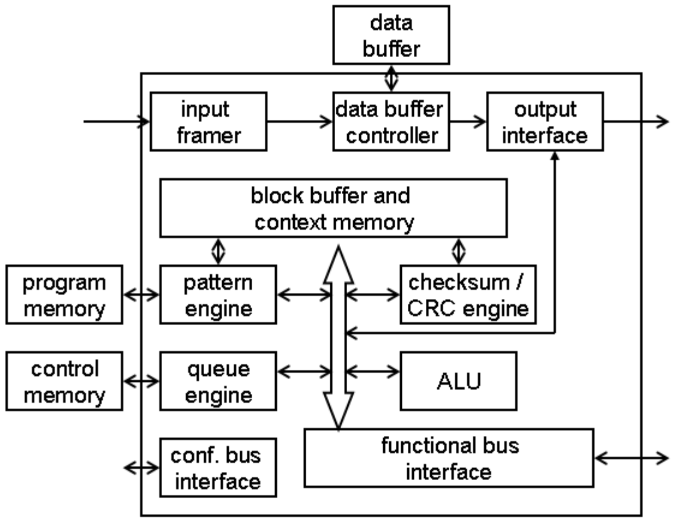


Figure 1.10: Internal structure of FPP unit.

are then forwarded toward the switching fabric. A third chip, the Agere System Interface (ASI) is a co-processor that introduces new functionalities to improve general performance. The ASI gathers statistical information on packets, which are then used for traffic management. Moreover the ASI provides a connection toward a distinct processor (not shown in figure), which is used to manage the overall system and the exception packets.
The system offers other connections: for instance, the configuration bus connects also the ingress hardware interfaces in order to coordinate the data flow toward the FPP. Fig. 1.10 shows the internal architecture of the FPP.

1.2.1.2 Processors and functional units

Each chip in the Agere system contains several processors and provides different functionalities. Tab 1.1 shows the features of each component. The Functional Bus Interface (FBI) implements an interesting form of Remote Procedure Call (RPC), which allows for calling functions which are external to the FPP unit. This way it is possible to extend the FPP functionalities by adding ASIC hardware.
Moreover, the FPP contains the interfaces for each external connection. For instance, the pattern processing engine can interface to an external control memory by means of a program memory and a queue engine. To handle packets, the FPP has an ingress interface, a framer that divides packets in 64 bit long blocks and an output interface for the configuration bus. The FPP contains also an external interface for the configuration bus. The central part is given by a functional bus, which all the processors can connect to. There is also an external interface for the functional bus

Table 1.1: Units and functionalities of Agere system.

Unit	Functionality
Pattern processing engine	Pattern matching
Queue engine	Manage packet queuing
Checksum/CRC engine	Compute checksum or CRC
ALU	Classical operations
Input interface and framer	Divide ingress packets in 64-byte long blocks
Data buffer controller	Check access to external data buffer
Configuration bus interface	Connect to external configuration
Functional bus interface	Connect to external functional bus
Output interface	Connect to external RSP chip

Table 1.2: Processors and functionalities of RSP unit.

Unit	Functionality
Stream editor engine	Packet modification
Traffic manager engine	Regulate traffic and hold statistics
Traffic shaper engine	Check QoS parameters
Input interface	Receive packets to FPP
Packet assembler	Store arriving packets
Queue manager logic	Interface to external traffic scheduler
Output interface	External connection for output packets
Configuration bus interface	Connect to external configuration bus

which is used by the ASI for checking the processing. The RSP unit, whom internal structure is shown in 1.11, has a set of processors and functionalities listed in tab. 1.2. The stream editor, the traffic manager and the traffic shaper have been built with Very Long Instruction Word (VLIW) processors.

1.2.1.3 Memory

In the Agere architecture, both external and internal memory are provided. The FPP divides packets in blocks and stores them in an external data-buffer (by means of an interface on the chip). It uses the internal memory for packets in the processing stage, while the external memory is used to store programs and instructions. The RSP stores packets in an external SDRAM and uses a Synchronous Static RAM (SSRAM) for high priority queues.

1.2.1.4 Programming support

Ease programming is an appealing feature of Agere chip. The FPP is a pipelined multithreaded processor and provides 64 independent contexts. However the parallelism is hidden to the programmer, who is able this way to use high-level languages.

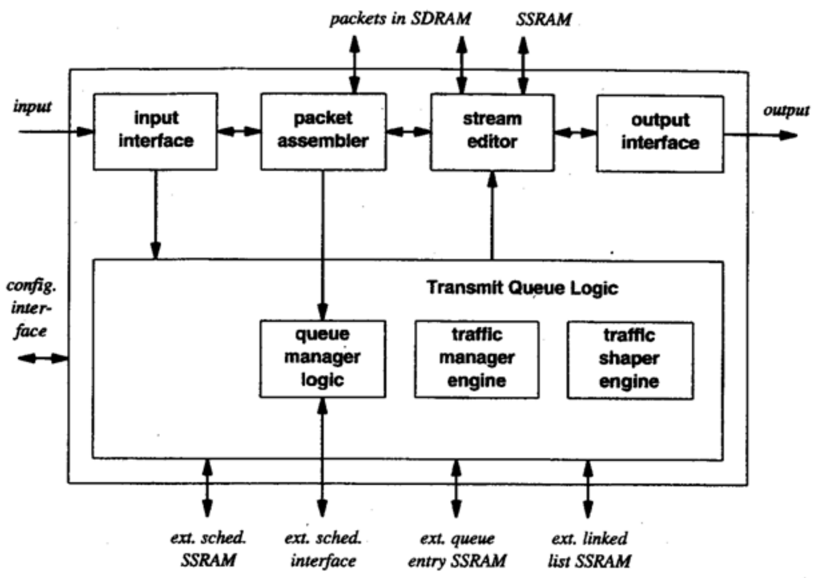


Figure 1.11: Internal structure of RSP.

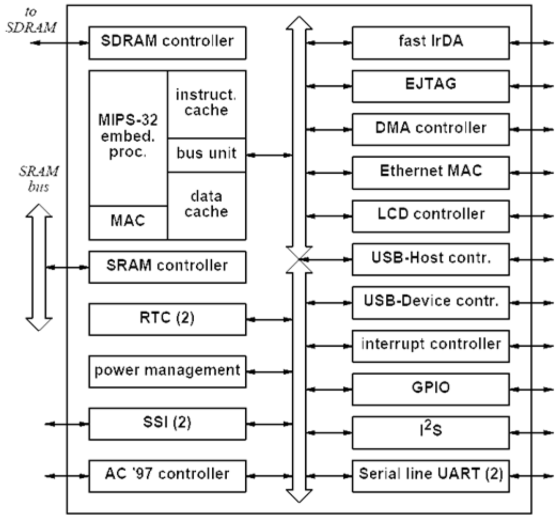


Figure 1.12: The Alchemy chip.

Agere offers also a specific language for the classification, a Functional Programming Language (FLP), and a scripting language, the ASL (Agere Scripting Language). Moreover, Agere offer a substantive support for traffic management. The logic of RSP allows for using multiple queues, applying external scheduling rules and handling traffic shaping.

1.2.2 Augmented RISC Processor (Alchemy)

Alchemy semiconductors Inc. (acquired by Advanced Micro Devices) offers different versions of Network Processors with different speeds [5]. These solutions are based on a RISC processor which is enriched by instructions specialized on packets processing.

1.2.2.1 Architecture

This architecture is characterized by an embedded RISC processor along with a series of co-processors. The core, a CPU MIPS-32, uses a 5-stages pipeline, the pipelined register file access and the zero penalty branching to improve the performance. Many instructions have been added to the set, such as a "multiply and accumulate" to aid in CRC or checksum computing. Other added instructions are those for memory prefetch, for conditional move operations, count leading of 0s and 1s. Fig. 1.12 shows the internal organization of Alchemy chip.

1.2.2.2 Processors and functional units

As shown in fig. 1.12, the embedded RISC processor can access to a certain number of I/O controllers and functional units. The chip contains also an RTC (Real Time Clock) unit.

1.2.2.3 Memory

On the chip there are two caches of 26KB, one for instructions and one for data, and connections for external SDRAM e SSRAM. The bus which connects the SSRAM provides also access to a Flash Memory, a ROM and a PCMCIA unit.

1.2.2.4 Programming support

Given that the Alchemy chip uses a MIPS processor, it can be programmed in C language.

1.2.3 Embedded Processor Plus Coprocessors (AMCC)

Applied Micro Circuit Corporation (AMCC) offers a series of NPs with different performance [6]. The AMCC architecture allows for efficiently using parallelism in order to obtain high data-rates.

1.2.3.1 Architecture

The version nP7510 includes 6 embedded processors (called nP cores), which work in parallel (e.g., a packet transform engine, a policy engine, a metering engine) and other functional units which provide external interfaces. An external co-processor handles address lookups based on a Longest Prefix Match algorithm. Fig. 1.13 shows the scheme of AMCC chip.

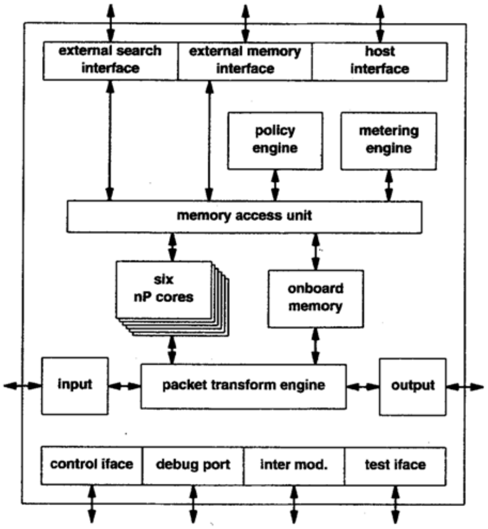


Figure 1.13: AMCC nP7510.

1.2.3.2 Processors and functional units

Each processor provides hardware threads at zero context switch. This way, the nP7510 can simultaneously process more packets or cells. The programming model of AMCC allows for hiding the parallelism to programmer, who can write code as for a single processor. Moreover, each packet or cell is processed by a single thread, this way avoiding to partition the code and implement complex balancing algorithms. The Packet Transform Engine, which is optimized for packets or cells, allows for operations on frames in parallel with the nP cores; several operations can be made in a single instruction: insert or delete data, compute and add the CRC or change values in packet header. The Special Purpose Engine enables the elimination of mutexes or other software threats for synchronizing access of thread to shared resources. The Policy Engine is dedicated to search and classification operations. Many lookups (up to 512 with compound keys) can be simultaneously made with a fixed latency. A key feature of Policy Engine is the "Network-Aware CASE Statement": the use of multiple and concurrent classifications allows for the elimination of nested "if-then-else" instructions, this way reducing code and improving performance. The metering engine enables the collection of information for the remote monitoring of SNMP, while the Statistic Engine enables the automated collection of statistics based on RMON protocol. The nP7510 has been designed to support a speed of 10 Gbps. It can be interfaced with the traffic management chipset nPX5710/20. The configuration can be doubled in order to handle a full duplex traffic of 10 Gbps. The nPX5710/20 contains also a virtual SAR unit (Segmentation And Reassembly).

1.2.3.3 Memory

As for many NPs, the AMCC chip offers external and internal memories. Moreover, a controller manages the two types of memory and hides this double nature to processor. An external TCAM is used for packet classification processes.

1.2.3.4 Programming support

These processors can be programmed in C or C++; AMCC provides a compiler, an assembler and a debugger.

1.2.4 Pipeline of Homogeneous Processors (Cisco)

The Parallel eXpress Forwarding (PXF) network processor has been designed by Cisco to be used in Cisco routers [7].

1.2.4.1 Architecture

The PXF adopts a parallel architecture that can be configured in order to create a series of pipelines. A single chip contains 16 embedded processors that can be put to work on 4 parallel pipelines. Figure 1.14 shows a possible organization of processors.

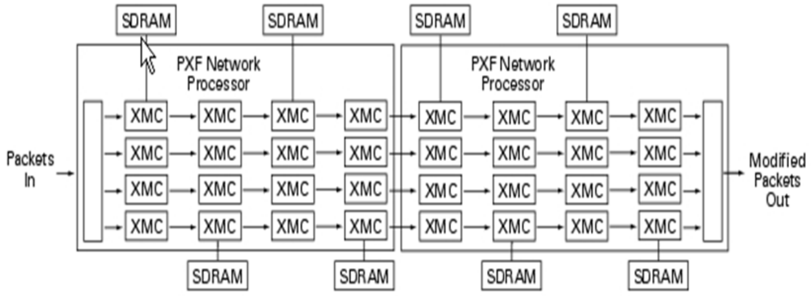


Figure 1.14: A possible configuration of CISCO XPF.

1.2.4.2 Processors and functional units

The PXF architecture counts a separation between control plane and forwarding plane. A route processor cares of routing protocols, network configuration, errors handling, and packets which are destined to the router. Instead, the forwarding plane is controlled by the PXF technology. In the PXF, each processor is optimized for packet processing at high speed and it is completely independent of the other ones; these units are called Express Micro Controllers (XMCs) and contain a complex double execution unit, provided with several specific instructions for an efficient packet processing. Moreover the XMCs can access to different resources on the chip, as register files and timers. They have also a shared access to an external memory in order to store state information, such as routing tables and packet queues. Finally, some micro-controllers guarantee that processing results can be passed among subsequent XMCs on the same pipeline. Figure 1.15 illustrates the path of a packet through this architecture. In this configuration, 2 PXF network processors are used for each Performance Routing Engine (PRE), this way obtaining 4 pipelines of 8 processors. Whenever a packet goes to a PRE from the ingress interface, it enters the ASIC backplane interface and is bufferized in the input packet memory. The header is extracted and sent to PXF for packet classification, header modification and, if needed, data modification. The processing comprehends also the selection of the port on which packet forwarding is performed. By means of simple routine algorithms, the PXF instructs ASIC backplane interface to store packet in its packet-buffer memory, in one of the possible queues which are associated to corresponding output queues. Then, the scheduling function of PXF processes this queue in order to determine what is the next packet to be forwarded. After this decision, the PXF instructs ASIC backplane interface to copy this packet in the hardware queue associated to corresponding egress interface.

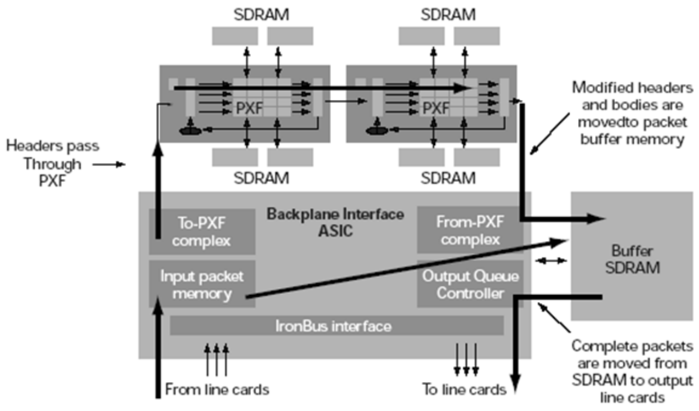


Figure 1.15: Standard path of a packet in a PRE.

1.2.4.3 Memory

There is an independent memory for each processor and one for each column of processors, in order to optimize accesses.

1.2.4.4 Programming support

This network processor is realized for internal use, and not as general-purpose product, thus it uses private software. Microcode and Cisco IOS are combined to provide processing functions. The association of these functions to the processors pipeline is very flexible and can be updated when new functions are available to be added.

1.2.5 Configurable Instruction Set (Cognigine)

The network processor of Cognigine Corporation is an example of reconfigurable logic: the adopted processor has not a prefixed set of instructions.

1.2.5.1 Architecture

This architecture allows for using up to 16 processors, which can be interconnected to form a pipeline. Each processor is called Reconfigurable Communication Unit (RCU) and has a connector that links it to RSF (Routing Switch Fabric), this way allowing for communications arbitrage and planning. The RCUs are connected in a hierarchical manner: a crossbar is used to connect a group of 4 RCUs and another one to connect groups of RCUs. This solution allows for scaling the architecture for a big number of RCUs. The RSF permits to divide a transaction in order to hide latencies; it is accessed by a RCU through a memory mapping. Each RCU contains 4 execution

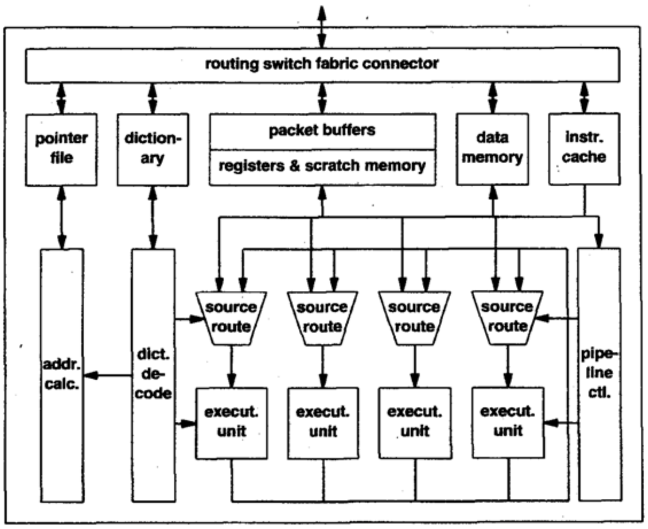


Figure 1.16: Internal structure of Cognigine network processor.

units which can be dynamically reconfigured. Each unit uses an instruction set called Variable Instruction Set Communications (VISC). As for a standard processor, a VISC instruction performs an easy operation, but details of operation are not determined a priori. In fact, the chip contains a dictionary which defines the interpretation of each instruction: operands' size, how they can be employed, the basic operation and the predicate. The dictionary is in turn configurable, elements can be added or dynamically changed. This way, programmer can define a personal instruction set, insert the interpretation of these instructions and develop a program based on them. For instance, a programmer could define an instruction set optimized for peculiar processings or specific protocols. VISC instructions are decoded during the first stage of the pipeline. Each RCU provides a five-stage pipeline and hardware support for 4 threads. The interconnections among processors are again configurable. For each RCU there are 4 64-bit data buses and 4 buses at 32-bit addresses, which allows for connecting RCUs in pipeline.

1.2.5.2 Memory

RCUs access to different types of memory, such as the internal SSRAM or the Double Data Rate SDRAM (DDR-SDRAM). Dictionary for VISC instructions is allotted in a distinct memory. Memories compose a hierarchy where the fastest ones are internal registers and scratchpad memory, then the cache for instructions and memory dedicated to data, while the slowest ones is the external memory, which is designed to store packets.

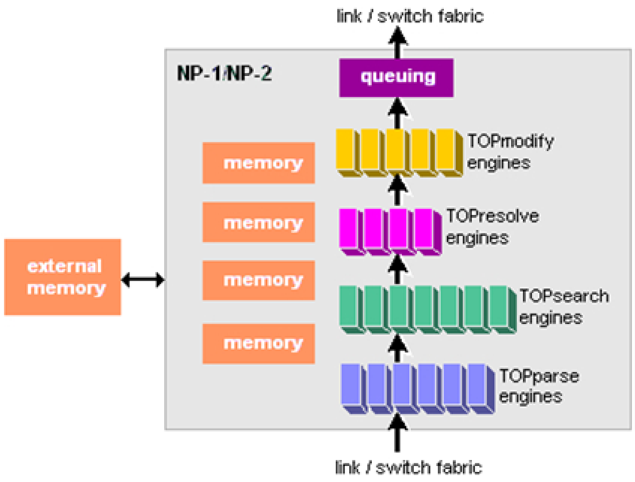


Figure 1.17: The scheme of NP-1 chip.

1.2.5.3 Programming support

In order to maximize the parallelism, the RCUs provide hardware support for multi-threading. Moreover, there are connections to external buses, as the PCI bus. Finally, along with C compiler and assembler, Cognigine offers a support for a classification language.

1.2.6 Pipeline of heterogeneous processors (EZchip)

EZchip Corporation produces the network processor NP-1 [8]. This architecture shows as heterogeneous processors, each of them dedicated to specific functions, can work together in a pipeline manner. The NP-1 has been designed for a big target: processing of layers 2-7 at 10 Gbps. This chip contains also a very fast SRAM, which is used for storing packets and lookup tables. There is an interface to access an external DRAM (external SRAMs and CAMs are not necessary). The chip includes also an interface for an external processor for management and control functions (the interface is not shown in figure). Moreover, EZchip claims to use patented algorithms which allows embedded memory for searching in external memories, in order to support a line-rate of 10 Gbps. These algorithms and the associated data structures allow for searches with strings of variable length. Further details are not publicly available.

1.2.6.1 Architecture

In this chip there are the Task Optimized Processors (TOPs). Each TOP has a personal set of instructions and connections which is specific for the functionalities that it must provide. Figure 1.17 illustrates chip architecture.
The NP-1 contains 4 types of processors, which are describer in tab. 1.3.

Table 1.3: Processors of NP-1.

Processor type	Optimized for
TOPparse	Header field extraction and classification
TOPsearch	Table lookup
TOPresolve	Queue management and forwarding
TOPmodify	Header and payload modification

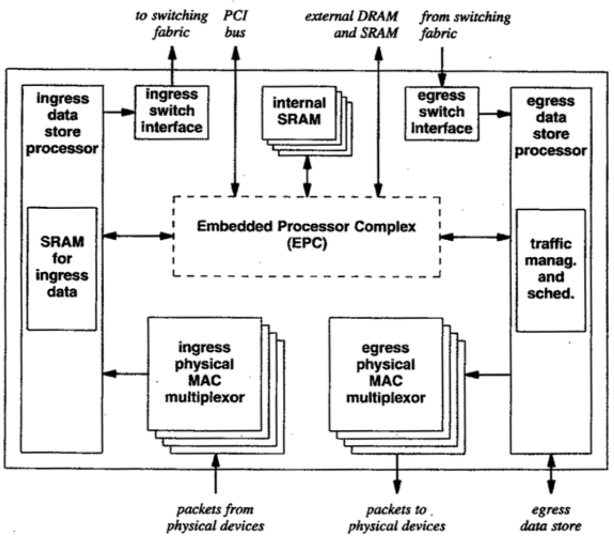

to switching fabric
PCI bus
external DRAM and SRAM
from switching fabric
ingress data store processor
ingress switch interface
internal SRAM
egress switch interface
egress data store processor
SRAM for ingress data
Embedded Processor Complex (EPC)
traffic manag. and sched.
ingress physical MAC multiplexor
egress physical MAC multiplexor
packets from physical devices
packets to physical devices
egress data store

Figure 1.18: Internal architecture of IBM network processor.

1.2.7 Extensive and Diverse Processors (IBM)

IBM produces a family of network processors called PowerNP [9]. This solution is very complex and comprehends a wide gamma of processors, co-processors and functional units.

1.2.7.1 Architecture

This network processor contains programmable processors and several co-processors which handle searches, frame forwarding, filtering and frame modification. The architecture is composed by a set of central embedded processors, along with many supporting units. Fig. 1.18 shows the overall architecture, while fig. 1.19 accurately illustrates the area called Embedded Processor Complex (EPC).
In addition to the embedded PowerPC, the EPC contains 16 programmable processors, which are called picoengines. Each picoengine is multithreaded, thus improving again performance. In order to speed up processing, frames are processed before being passed to the protocol processor in the EPC. The ingress physical MAC multiplexor

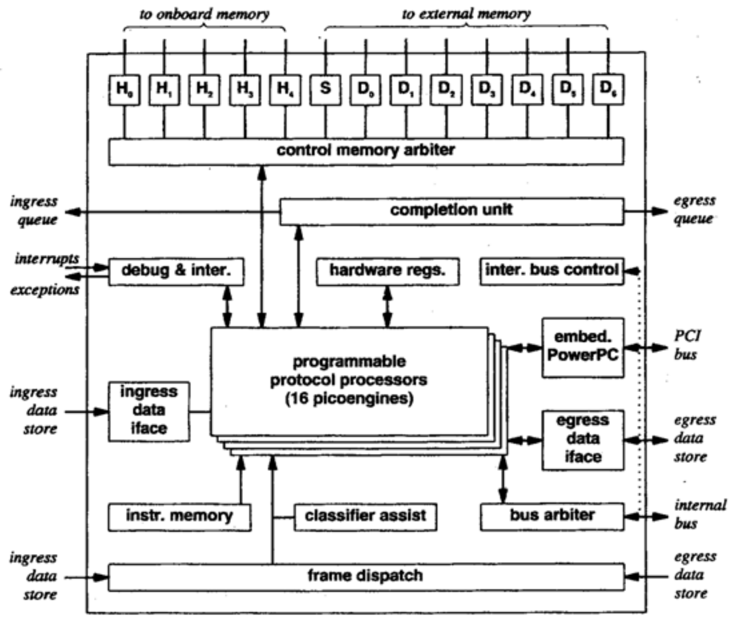


Figure 1.19: The EPC chip in the IBM NP.

Table 1.4: Co-processors of IBM NP.

Co-processor	Function
Data Store	Frame buffer DMA
Checksum	Compute and check header checksums
Enqueue	Forward frames arriving from switch or target queues
Interface	Provide access to internal registers and memory
String Copy	Transfer big amounts of data at high speed
Counter	Update counters used in protocol processing
Policy	Handle traffic
Semaphore	Coordinate and synchronize threads

takes frames arriving from physical interface, checks CRC and passes frames to ingress data store. The first part of frame, which contains headers up to layer 4, is passed to the protocol processors, while the remaining part is stored in memory. Once frame has been elaborated, the ingress switch interface forwards it toward the proper output processor through the switching fabric. The external hardware of the EPC takes care also of the output of frames. The egress switch interface receives data from the switching fabric and stores them in the egress data store. The egress physical MAC multiplexor handles frame transmission, by extracting them from egress data store and sending them to physical interface. In addition to picoengines, the chip of IBM contains several co-processors specialized for particular functions. Some examples are presented in table 1.4.

1.2.7.2 Memory

The PowerNP provides access to an external DDR-SDRAM and presents many internal memories, with an arbiter which coordinates accesses to them. The internal SRAM provides fast access, which allows for temporarily storing packets to be processed. Moreover, programmable processors have a dedicated instruction memory; for instance, each picoengine has 128 KB of private memory which is dedicated to this purpose.

1.2.7.3 Programming support

In addition to standard programming tool (such as compilers, assemblers, etc.), the IBM chip provides a software package for simulation and debugging. This package is available for several operative systems, such as Solaris, Linux and Windows. The co-processor that cares about traffic management works at wire speed, this way the IBM chip is able to analyze each packet in order to verify that traffic is complying to predetermined parameters.

1.2.8 Flexible RISC Plus Coprocessors (Motorola)

The Motorola Corporation brands its network processors C-Port. Models C-5, C-5e and C-3 represent a tradeoff between performance and power consumption.

1.2.8.1 Architecture

The Motorola chip is very appealing; it is an example of internal processors which can be configured to work in a parallel or pipeline manner. The capability of selecting a configuration model for each processor provides a high flexibility to C-Port. Fig. 1.20 shows as C-Ports can connect more physical interfaces to a switching fabric. Each network processor includes 16 blocks of processors, which are called Channel Processors and care for packet processing. Each CP can be configured in different ways. The most direct approach is the dedicated configuration, which establishes a one-to-one relation between the CP and the physical interface. In this configuration the Channel Processor must manage both the input and the output, and is suitable for interfaces at medium or low speed (100Base-T Ethernet or OC-3), for which the processor has enough power. In order to handle higher speeds, the Channel Processors

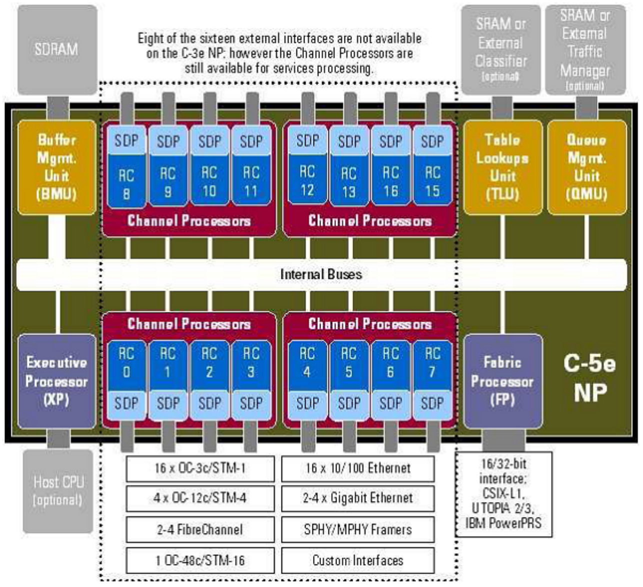


Figure 1.20: Architecture of C-Port.

can be organized in a cluster in a parallel way. This way, whenever a packet arrives, any CP in idle state can handle such a packet. The number of CPs in each cluster can be modified, thus the designer can select the proper sizes according to the interface speeds and the amount of required processing. Figure 1.20 shows chip C-Port C-5 architecture, where CPs are configured in cluster. The diagram illustrates the 16 Channel Processors ($CP-0 \ldots CP-15$) configured in parallel clusters of 4 CPs per cluster. In addition to CPs, the Motorola chip contains many other co-processors. The Executive Processor provides a configuration and management service of the overall Network Processor; it communicates with a potential host PC via bus PCI or through serial lines. The Fabric processor allows for a fast connection between the internal buses and an external switching fabric. The lookup unit allows for speeding up searches in lookup tables. The buffer management and queue management units handle and check respectively buffers for packets and queues. However, the name Channel Processor is misleading: the chip does not contains an only processor, but is a complex structure with a RISC processor and several functional units which aid in handling packets at high speed. Fig. 1.21 shows CP components and their interconnections. As we see, the CP has a parallel structure for ingress and egress side. The Serial Data Processor (SDP) is programmable and on the ingress side cares for checking checksum or CRC, decoding, analyzing headers, while on the egress side is used for modifying frames, computing checksum or CRC, coding, and framing. The RISC processor deals with classification processes, traffic handle and traffic shaping.

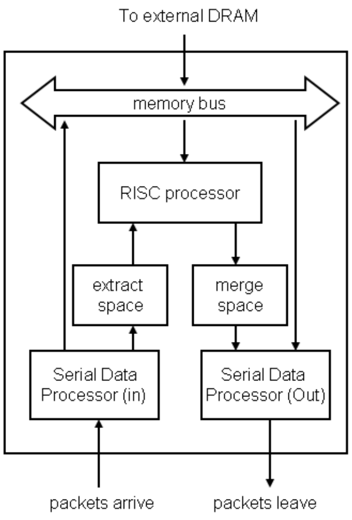


Figure 1.21: Internal architecture of a Channel Processor.

1.2.8.2 Programming support

The network processor C-Port can be programmed in C o C++. Motorola provides a compiler, a simulator, APIs and libraries to be used for managing physical interfaces, lookup tables, buffers, and queues.

1.3 Intel IXP2XXX Network Processors

In this section, the architecture and the functionalities of Intel IXP2XXX Network Processors will be shown. The characters XXX indicate the ciphers which specifies a particular model. We will refer to the overall family; the differences among models are related to the number of processing units, or the availability of specific functionalities (for instance, units which allow for encryption algorithms). Therefore, we try to explain the main features of IXP2XXX family, its advanced functions, programming languages, and develop environment. Finally the card Radisys ENP-2611 we have used will be described, which contains the Intel chip.

1.3.1 General Structure

Fig. 1.22 shows a scheme of the IXP2400, in which functional units and connections are presented. Often we refer to IXP2400 for specific features and data we give. The network processor contains 9 programmable processors: an Intel XScale and 8 units called microengine, which are divided in 2 cluster of 4 microengines (ME 0:0 . . . ME 1:3). The general purpose processor XScale is a RISC (Reduced Instruction Set Computer) ARM V5STE compliant, while the microengines are RISCs optimized for packet processing. From the scheme in fig. 1.22 is clear the use of memories with different sizes and features (e.g., SRAM, DRAM, Scratchpad), as well as the

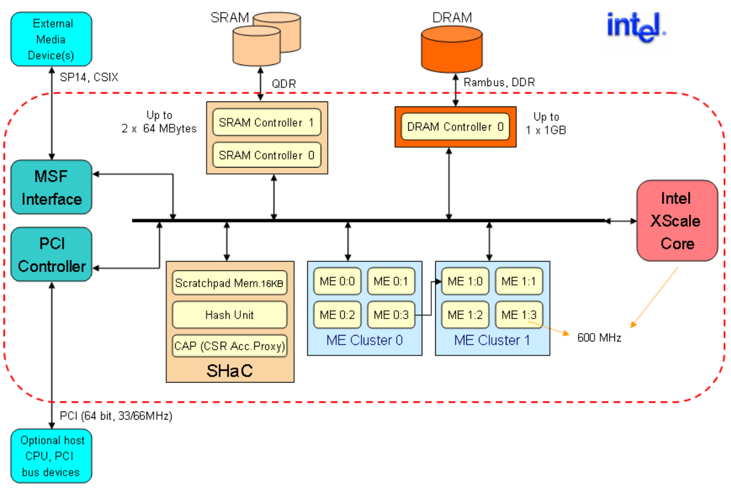


Figure 1.22: Scheme of the IXP2400.

availability of shared functional units with specific purposes (e.g., MSF or the unit for hash computing). In the following, all these features will be analyzed.

1.3.2 The Intel XScale

The Intel XScale processor which is installed on network processor of Intel family IXP2XXX is compliant with the ARMv5STE architecture, as defined by ARM Limited. The "T" indicates the support to thumb instructions, i.e. specific instructions which allow for passing from the 32bit modality to the 16bit one, and vice versa. This capability is useful for memory utilization purposes. Instead, the "E" indicates the support to advanced instructions of Digital Signal Processing. The processor uses an advanced internal pipeline, which improves the capability of hiding memory latencies. The support to floating point operations is not available.
Regarding the programming, the XScale processor supports real time operative systems for embedded systems as VxWorks or Linux. Therefore, it can take advantage of C/C++ compilers available in this environments. In addition, it can use several development tools, as IDE (Integrated Development Environment), and debuggers. In the IXP2400 NP, the XScale runs at 600 Mhz, while in the IXP2350 it runs at 1.2 Ghz.

1.3.3 Microengines

Microengines has a specific instruction set for processing packets. There are 50 different instructions, including the operations concerning the ALU (Aritmetic Logical Unit) which work on bits, bytes and longwords and can introduce shift or rotations in a single operation. The support to divisions or floating point operations is not available. The microengines of IXP2400 work at 600 Mhz, instead those of IXP2350 work at 900 Mhz or 1.2 Ghz. The memory which stores the code to be executed in a microengine is the instruction store and can contains up to 4K of 40bit instructions. The code is loaded on microengines by XScale processor in the startup phase. Once microengines runs, the instructions are executed in a 6-stage pipeline, requiring a clock cycle with full pipeline. Clearly, whenever jumps or context swaps happen, the pipeline must be cleared out and then filled again with instructions, thus way requiring more clock cycles.

1.3.3.1 Threads

Each microengine allows for the use of 8 thread with hardware support to context switch. This way of context switch is called "zero-overhead", because microengines hold a series of registers for each thread; thus, whenever the context switch occurs, registers copy is not required, therefore the overhead is related only to the pipeline emptying (i.e., very few clock cycles). Processors can be configured to use 8 threads, or only 4 threads. In the latter case, only the threads with even index are activated and they have a higher number of registers. All the threads execute the same instructions, which have been read from the internal memory of microengines, by starting from the first instruction. However, it is possible to differentiate the operations for each thread by using some conditional instructions:

```
if (ctx==1) {
...
}
else if (ctx==2) {
...
}
```

Each thread runs and then releases the controller to allows the other ones to run. The scheduling is not preemptive: until a threads works and does not release the controller, the other threads can not execute their code. The context switch is invoked by means of proper instructions (ctx_arb) and is typically used as mechanism for hiding access latency to resources. For instance, whenever an external memory must be read, the thread release the controller before it accessing to the memory. The not preemptive approach allows for reducing issues in critical sections, i.e. parts of code in which resources which are global for threads are used and modified. If two threads access to the same register at the same time, the data in the register can become insubstantial. Therefore, the not preemptive model aid in this purpose. However, the not preemptive scheduling does not solve the issue of critical sections for threads accessing contemporaneously to the same resource and belonging to different threads. Techniques of synchronization are therefore needed. To handle the threads execution for each microengine there is a thread arbiter, i.e. a scheduler which selects the thread to run by using a round-robin policy among the active threads.

1.3.3.2 Registers

There are four types of registers for each microengine:

- general purpose;
- SRAM transfer;
- DRAM transfer;
- next-neighbor.

As said above, each context has a private set of registers, therefore each bank of registers is divided in the same way among threads. In addition, there are some control Status Registers (CSRs) which allows for different operations or for configuring microengines' functioning. *General Purpose Registers (GPRs)* - Each microengine have 256 32-bit registers for general purpose, which are allotted in two banks of 128 registers (called bank A and bank B). Each instruction which has as operands GPRs, requires that they belong to different register banks. Registers can be accessed in local manner for the thread (i.e., each thread accesses 32 GPRs), or in absolute manner, or in global manner (i.e., registers are accessed by all the threads as global variables). In the code, name of GPRs can follow some rules [10]. *Transfer Registers* - SRAM transfer registers (256 per microengine) are 32-bit registers which are used for writing and reading from SRAM memory or from the other memories or functional units in the Network Processor, such as Scratchpad memory, SHaC unit, Media Switch Fabric, and PCI (Peripheral Component Interconnects) interfaces. DRAM transfer registers are suitable for writing ad reading from DRAM and can be used in replacement of SRAM registers only for reading. Transfer registers are the main mechanism to make asynchronous operations on the memories; on a transfer register a thread writes data to be then written in memory, or from a transfer register a thread reads data which has just been read from memory. Registers' bank is divided into two parts, one of them for writing and the other one for reading. This does not allow a wrong use of transfer registers (for instance, as GPRs). More precisely, when a transfer register is used, typically a couple of registers is available, with the same name, but writing on this register means writing on the "writing" part, while reading it means accessing the "reading" part. Also these registers can be accessed in local or global manner respecting to thread. *Next-Neighbor Registers* - Each microengine has 128 32-bit registers called next-neighbors. They can be used in two ways: as other general purpose registers, or as "microengine communication" registers. In the first case, if the standard general purpose registers are finished, for instance, the next-neighbor registers can be used in replacement. In the second one, they make available to the microengine with the next index the data which has just been written by the current microengine. This way, the first microengine can communicate with the second one, the second one with the third one and so on. The communication can occur through a simple writing in the registers or through the set up on the registers of a data structure called ring, which is a FIFO queue and which is accessed by means two CSRs, NN_PUT ans NN_GET.

1.3.3.3 Signaling

Each microengine has on tap 15 numbered signals. They are useful for the execution of asynchronous operations which concern memories and functional unit. For instance, whenever a reading in SRAM is required by a thread, the end of the operation can be

Table 1.5: Properties of IXP2400 memories.

Memory	Logical Width (bytes)	Size (bytes)	Latency (clock cycles)
Local Memory	4	2560	~ 3
Scratchpad	4	16k	~ 60
SRAM	4	128M	~ 90
DRAM	8	1G	~ 120

communicated through a signal to the thread which have required the reading. Once the signal from the SRAM has been received, we can be sure to have the data. Some functional units, for instance DRAM, require the use of a couple of signal for the signaling. Specific instructions allows for making context-switch and waiting for the arrival of one or more signals. This way the mechanism of hiding memory latency is obtained. Finally, signals can be used as synchronization mechanism among threads, in order to solve potential collisions on the same resources.

1.3.3.4 Local Memory

The local memory of a microengine consists of 640 longwords (i.e., words of 32 bits) which can be accessed very fast, with a maximum latency of 3 clock cycles. Moreover, this delay has to be taken in account only a specific CSR is used to select the position where we must work; if we use consecutive locations of local memory, we do not need to set again the CSR and then to wait for 3 other cycles. The access occurs through the special registers *l$index0 and *l$index1, which refer two different locations in local memory. Such registers, which are replicated for each thread, can be incremented or decremented (e.g., *l$index++) or used with indexes (e.g., *l$index[4] indicates the fourth longword after that indicated by *l$index[0]).

1.3.3.5 Content-Addressable Memory and CRC

The Content Addressable Memory (CAM) is a special memory which is addressable according to the content. Each microengine has a CAM with 16 entry. Specific instructions (CAM_LOOKUP) allow for search of a particular content on the memory. If the content is found, the CAM position is returned, otherwise the least recently used (LRU) element. The CAM is very useful to implement little cache or to handle arrays of queues. Finally, computing CRC (Cyclic Redundancy Check) is possible through proper registers.

1.3.4 Memories

IXP2XXX network processors can access 4 different types of memory: local memory, scratchpad, SRAM, and DRAM. The local memory can be accessed only by the single microengine that contains it, while the other memories are shared. Tab. 1.5 shows the different characteristics and tradeoff in terms of size, latency and minimum accessible unit (logical width).

Each type of memory allows for special operations. We have already said about local memory. The scratchpad is a SRAM memory on the chip, which is contained in the SHaC block. It allows for atomic operations on data, such as increase, decrease, test&set. An atomic operation in an operation that can not be divided. For instance, incrementing a variable requires reading, incrementing and writing it. If the overall operation is atomic, the three operations can be split. This way, the collision issues in the use of shared resources are solved. Moreover, the scratchpad enables creation and management of FIFO queues (which are called Rings, because they use a part of memory as it was circular). These "ScratchRings" are often utilized in order to permit the communication among microengines through simple and fast operations (the scratchpad is the fastest memory shared among microengines). The SRAM is an external memory which supports the same operations of the scratchpad; in additions, it allows for creating and handling FIFO queue by means of element pointers, therefore with no need to transfer them. no need to transfer them. The DRAM is the biggest and slowest memory. It allows for a direct path from and toward Media Switch Fabric with no need of transfer through microengines. The logical width has to be taken in account in programming phase, because mechanisms to hide it to the programmer are not available. This way, for example, to access two consecutive longwords in SRAM, we need to indicate the second one with an offset of 4 in respect to the first one. Finally, it is useful to know the management of asynchronous commands on the shared memories which arrive from different threads. Each interface of the memories has a queue of command to be executed, from which draw on in a sequential manner. A thread can have more command on different queues or on the same queue.

1.3.5 Media Switch Fabric

The Media Switch Fabric (MSF) unit is the interface designed to data transfer from and toward network processors of IXP2XXX family. Packet reception and transmission on network processors is a complex process of reassembly and segmentation of little parts of packets called mpackets. Through MSF, programmer has an interface for transmission and reception which is universal and independent of packet format. The mpacket size is defined by the reception buffer (RBUF) and by the transmission buffer (TBUF), which are configurable in 64, 128 and 256 bytes through specific CSRs of MSF.

1.3.6 SHaC

SHaC (Scratchpad, Hash and CAP) is the multifunction unit which contains the scratchpad memory, an unit for generation of hash codes and the CAP (Control Status Register Access Proxy) unit. The hash unit is capable of computing hash codes of 48, 64 and 128 bits from keys of the same size. Moreover, with an only request, 3 keys to be worked on can be inserted. The algorithm to be used can be configured through CAP. The CAP unit provides the interface for using many CSRs for the overall chip. In addition, it allows for the inter-threads and inter-microengines signaling and the management of interrupts to be sent to XScale processor. Another functionality of CAP is the handling of register reflector, which is a mechanism used by a thread in a microengine in order to write on the SRAM transfer registers of any other thread in any microengine. Finally, the SHaC contains also the logic for interfacing the peripherals of XScale processor as memories and external timers.

1.3.7 Intel IXA Portability Framework

Intel Internet Exchange Architecture (IXA) takes care of providing hardware which is programmable via software and open APIs. Practically, it is the hardware and software architecture of Intel network processor family. The IXA Portability Framework is the modular architecture which is based on building blocks and allows for the reuse of the code written for a IXP2XXX NP on any NP of the same family. Therefore, the software structure is based on the modular modality of code for XScale and microengines which is supported by an Hardware Abstraction Layer with standard APIs. The flexibility is guaranteed through the full programmability of the two architecture layers and the different low-level functions which are provided in hardware. Moreover, it is possible to select the model of multithreaded programming (parallel way, pipeline or hybrid) according to the needs. In addition, hardware which is expressly designed for the IXA architecture permits to solve the issues of memory latencies, which raise when the rate grows.

1.3.7.1 Microblocks and Core Components

The modular structure of the software, which enables the code portability, is based on two types of building blocks. They are called *core components* at XScale level and *microblocks* at microengines level. Each building block represents a functionality of packet processing, e.g. NAT, forwarding, Ethernet bridging, etc. Programmer can use these elements or build new ones or combine them to create an application. Some blocks are called *driver-blocks* and care about the operations more dependent by the underlying hardware architecture, such as reception, transmission or queue handling. They are blocks optimized for their purposes, therefore it is not opportune to modify them.

1.3.7.2 XScale/microengines interactions

The network processors of IXP2XXX family present two hierarchical layers:

- an upper layer, with the XScale processor (programmable in C language), which hosts an embedded operative system and deals with control plane and management of the overall NP;
- a lower layer, which takes care of fast data path and is composed of microengine (programmable in microcode assembly), which provide a short set of instructions optimize for packet processing.

The core components operate as intermediate between these two layers. They are modules which allows for the interface between the processor and all the other units of NP, for defining symbols and for handling exceptions of fast data path. The use of symbols is useful for the definition of resources. Indeed, some modules care about resource management in an integrated manner, i.e. each use of any memory part requires a direct request to a module called *resource manager*. The resource manager, in the process of microcode loading on the microengines, will allocate the required resources and will set the proper parameters for the application functioning (this phase is called *symbol* patching). Each block of code written for a microengine can be handled also by a core component at XScale level. Therefore there is a core component for the reception code, another one for the transmission code, and so on.

1.3.8 IXA SDK

In addition to the IXA Portability Framework, the Intel IXA SDK (Software Development Kit) provides several tools to develop applications for IXP2XXX NPs. These tools include a compiler for a microengine-oriented C-like language, an assembler for the assembly language for microengines [11] and a Integrated Development Environment (IDE) called Developer's Workbench.

1.3.8.1 Assembly for microengines

The instruction for the assembly language for microengine [12] assume this general form:

opcode [param1, param2, ...], opt1, opt2, ...

With *opcode* we indicate the name of instruction, the parameters to be passed are *param1*, *param2*, etc., and there are also the optional parameters *opt1*, *opt2*, etc. These options attend to change the behavior of the instruction or to add optimizations. For instance, a common option is *ctx_swap[signal]*: it allows, in instructions which access memory, for executing a context switch by waiting for a signal from memory controller, which points that data have been read or written. Other common options allows for code optimization, by reducing penalty in case of jump. These options are *defer[n]*, which point to the assembler to execute the first n next instructions in the pipeline before a jump or a context switch. The assembly language for microengine gives the possibility of conditioned or not-conditioned jumps, as well as any other programming language. The points on the code to which jump are indicated through labels followed by the character #. For instance

label1#

...

...

br[label1#]

In the instruction set, the opcodes point typically the hardware unit to be used. For example, if two registers have to be summed, the instructions is:

alu[z,x,+,y]

because for the arithmetical logical operations the ALU (Arithmetic Logic Unit) is used. Instead, if a reading in SRAM is required, the following instruction is used:

sram[read,x,position,0,1],ctx_swap[sig_sram]

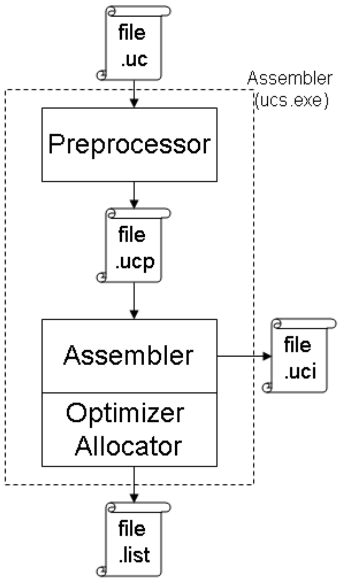


Figure 1.23: Compilation process.

1.3.8.2 Constructs

The assembler provides some user-friendly constructs, which replicate the basic constructs of the most widespread programming languages. Thus, *if endif*, *while*, *repeat until*, can be used. This way, code is more readable and less prone to wrongs. Moreover, it is possible to create subroutines to be called, but commonly they are not utilized because the stack lacks. Instead, macros are preferred, i.e. code which is exploded for each occurrence. Macros, along with conditional compilation and other functions, are made possible by a preprocessor, very similar to the preprocessor of C language, which is a very useful tool for programming in assembly.
The overall compilation process is shown in fig. 1.23: we start from the .uc file to arrive to the .list file, which contains the actual code to be executed by a microengine.

1.3.8.3 Virtual registers

The assembler provides the capability to handle the available registers through some names, although a name of a register does not point always the same location in the registers banks. These are the *virtual registers*, which allows for defining different scopes for registers. For instance, let us suppose that a macro for the debug utilizes a couple of registers, which are then never used in the remaining part of the code. It should be a wastage to statically allocate two locations in the banks for these two registers. Therefore, the key-words *.begin* and *.end* are used, this way defining a scope for the registers: out of this scope, the registers do not exist and the corresponding memory locations can be reused. The mechanism of dynamical mapping of registers on physical locations is not only related to the functions in order to define the scope.

In fact, if the number of declared registers raises so much that they can not be all statically allocated, some physical locations used by a certain register (with a scope still active) are reused and then assigned again to the original register when it needs them. This mechanism can be dangerous if used on transfer registers which are currently used for accessing memories. For this case, there is the key word *volatile* which guarantees the statical allocation of registers.

1.3.8.4 Microengine-C

The microengine-C language allows for programming microengines with the ease and the typical features of C language, i.e. types check, memory pointers and functions. Since a memory stack can not be used, functions defined in microengine-C can not be recursively called and functions' pointers can be used. The syntax of microengine-C is compliant to ANSI-C, except these limitations concerning functions. The supported types are *signed* and *unsigned* and go from *char* (8 bits) to *longlong* (64 bits). Moreover, *structs* and *enum* types are supported. According to the optimizations of compiler, functions can be compiled as online (i.e., they are exploded as they are macros) or as subroutines. Given the different type of memory and registers, declarations of variables must be accompanied by indications regarding their allocation. Moreover, some specific functions allowed by the NP, such as atomic operations, have not a corresponding one in ANSI-C. Therefore, "intrinsics" are used, which are constructs expressly introduced, which look as functions but in actuality correspond to well know sequences in assembly. These differences from the common C make the use of microengine-C less easy and perceptual. In addition, the compilation process does not allow for obtaining optimized code, so it pass through an assembly version. For these reasons, often the assembly is preferred to microengine-C.

1.3.9 Developer's Workbench

The Integrated Development Environment provided with IXA SDK is the Developer's Workbench. This development tool allows for writing code and debugging of assembly or microengine-C in a visual envornment in Windows Microsoft. Moreover, it is possible to debug the code in hardware, by connecting the Network Processor to the PC with the IXA SDK. Finally, an accurate simulator of Network Processor (based on clock cycles and not event-driven) is provided as part of IDE. It precisely recreates system behavior and is an optimum tool for testing applications' prototypes with no need to port the code on the hardware and for the accurate measurements, for example, of latencies of single processing stages in the network processor.

1.3.9.1 Scripting

The simulator of Developer's Workbench (called Transactor) supports a C-like scripting language. It provides several commands which permit to accurately observe applications behavior. Indeed instructions to add a *watch* on memories and registers are available, with the capability to execute specific sequences of instructions when certain values change. For example, each time a register changes its value, the content can be written on a file. The values of registers or memory locations can be initialized or modified, the RBUF and TBUF buffers can be obseved, as well as CSRs of any block of network processor. The definition and the use of functions is supported, as well as the the use of classical constructs of programming languages, such as if(), while(), etc. Finally, there are further commands for the simulation control, i.e. model reset, simulation stop or restart, and so on.

1.3.10 ENP-2611

Laboratories which have placed this research have on tap Radisys ENP-2611 cards, on which is integrated the Network Processor Intel IXP2400. These medium-low profile cards allow for obtaining nominal line rates of 2.5 Gbps and have 3 optical multimodal gigabit ethernet ports. A further gigabit port at 10/100 Mbps is available in order to handle traffic of control plane or for debugging services. These cards are mounted on PC through a PCI bus compliant with the specifications 2.2 at 32 or 64 bits. The use of PCI bus permits to connect more ENP-2611 cards in order to build a single network node. These cards provide 8 Mbytes of SRAM and 256 Mbytes of DRAM. The development on ENP-2611 cards is based on IXA SDK, but Radisys has introduced an own add-on to SDK Intel, which is called ENP-SDK.

Chapter 2

Deep Packet Inspection

Many modern network devices need to perform Deep Packet Inspection (DPI) at high speed, in order to improve network security and provide application-specific services. The deep packet inspection problem is basically the inspection of traffic in order to look for the occurrence of particular strings of "patterns" of bytes into packet payloads. This is very useful to classify traffic up to the top layer of the network protocol stack. Instead of standard strings to represent the data set to be matched, state-of-the-art systems use regular expressions, due to their high expressive power and flexibility. Typically, finite automata (FAs) are employed to implement regular expression search, but for the current string sets they need a memory amount which turns out to be too large for practical implementation. Many recent works have proposed improvements to address this issue. They are adopted by well known IDS tools, such as Snort [13] and Bro [14], and in firewalls and devices by different vendors such as Cisco[15]. However, Finite Automata suffer from either speed issues (if they are non deterministic) or size ones (if, on the contrary, they allow deterministic lookups). In this chapter we describe a number of solutions for both problems. Unfortunately, standard pattern matching methods used for deep packet inspection and network security can be evaded by means of TCP and IP fragmentation. In order to detect such attacks, Intrusion Detection Systems must reassemble packets before applying matching algorithms, thus requiring huge memory and a large amount of time to respond to the threat. For this reason, we introduce also an efficient system for anti-evasion, which can be implemented in real devices. It is based on Counting Bloom Filters and exploits their capabilities to quickly update the string set and deal with partial signatures. In this way, almost all the traffic processing is performed in the fast data path, as well as the attacks detection, thus improving the scalability of Intrusion Detection Systems. A brief introduction to Hash and Bloom Filters could be found in appendix A.1 and in appendix A.2.

2.1 δFA: An Improved DFA construction for fast and efficient regular expression matching

Many important services in current networks are based on payload inspection, in addition to headers processing. Intrusion Detection/Prevention Systems as well as traffic monitoring and layer-7 filtering require an accurate analysis of packet content in search of matching with a predefined data set of patterns. Such patterns characterize specific classes of applications, viruses or protocol definitions, and are continuously updated. Traditionally, the data sets were constituted of a number of signatures to be searched with string matching algorithms, but nowadays regular expressions are used, due to their increased expressiveness and ability to describe a wide variety of payload signatures [16]. They are adopted by well known tools, such as Snort [13] and Bro [14], and in firewalls and devices by different vendors such as Cisco[15]. Typically, finite automata are employed to implement regular expression matching. Nondeterministic FAs (NFAs) are representations which require more state transitions per character, thus having a time complexity for lookup of $O(m)$, where m is the number of states in the NFA; on the other hand, they are very space-efficient structures. Instead, Deterministic FAs (DFAs) require only one state traversal per character, but for the current regular expression sets they need an excessive amount of memory. For these reasons, such solutions do not seem to be proper for implementation in real deep packet inspection devices, which require to perform on line packet processing at high speeds. Therefore, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regular expression sets [17][18][19][20]. This work focuses in memory savings for DFAs, by introducing a novel compact representation scheme (named δFA) which is based on the observation that, since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only. The δ in δFA just emphasizes that it focuses on the differences between adjacent states. Reducing the redundancy of transitions appears to be very appealing, since the recent general trend in the proposals for compact and fast DFAs construction (see sec.2.1.1) suggests that the information should be moved towards edges rather than states. Our idea comes from D^{2}FA [17], which introduces default transitions (and a "path delay") for this purpose. Unlike the other proposed algorithms, this scheme examines one state per character only, thus reducing the number of memory accesses and speeding up the overall lookup process. Moreover, it is ortoghonal to several previous algorithms (even the most recent XFAs [20][21][22] and H-cFA [18]), thus allowing for higher compression rates. Finally, a new encoding scheme for states is proposed (which we will refer to as *Char-State compression*), which exploits the association of many states with a few input characters. Such a compression scheme can be efficiently integrated into the δFA algorithm, allowing a further memory reduction with a negligible increase in the state lookup time. In summary, the main contributions of this work are:

- a novel compact representation of DFA states (δFA) which allows for iterative reduction of the number of states and for faster string matching;

- a new state encoding scheme (*Char-State compression*) based on input characters;

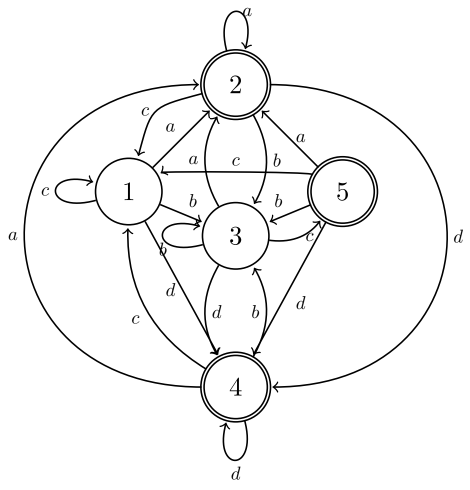
The remainder of the section is organized as follows. In section 2.1.1 related works about pattern matching and DFAs are discussed. Sec.2.1.2 describes our algorithm, by starting from a motivating example and sec.2.1.3 proves the integration of our scheme with the previous ones. Then in sec.2.1.4 the encoding scheme for states is illustrated and in the subsequent section the integration with δFA is shown. Finally, sec.2.1.7 presents the experimental results.

2.1.1 Related Work

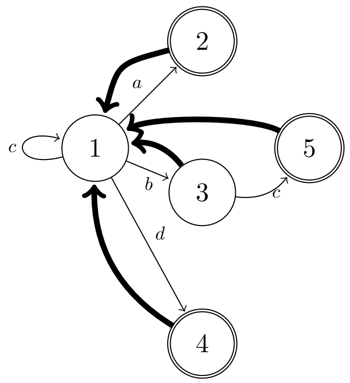
Deep packet inspection consists of processing the entire packet payload and identifying a set of predefined patterns. Many algorithms of standard pattern matching have been proposed [23][24][25], and also several improvements to them. In [26] the authors apply two techniques to Aho-Corasick algorithm to reduce its memory consumption. In details, by borrowing an idea from Eatherton's Tree Bitmap [27], they use a bitmap to compress the space near the root of the state machine, where the nodes are very dense, while path compressed nodes and failure pointers are exploited for the remaining space, where the nodes become long sequential strings with only one next state each. Nowadays, state-of-the-art systems replace string sets with regular expressions, due to their superior expressive power and flexibility, as first shown in [16]. Typically, regular expressions are searched through DFAs, which have appealing features, such as one transition for each character, which means a fixed number of memory accesses. However, it has been proved that DFAs corresponding to a large set of regular expressions can blow up in space, and many recent works have been presented with the aim of reducing their memory footprint. In [28] the authors develop a grouping scheme that can strategically compile a set of regular expressions into several DFAs evaluated by different engines, resulting in a space decrease, while the required memory bandwidth linearly increases with the number of active engines. In [17], Kumar et al. introduce the Delayed Input DFA (D^2FA), a new representation which reduces space requirements, by retrieving an idea illustrated in [29]. Since many states have similar sets of outgoing transitions, redundant transitions can be replaced with a single default one, this way obtaining a reduction of more than 95%. The drawback of this approach is the traversal of multiple states when processing a single input character, which entails a memory bandwidth increase to evaluate regular expressions. To address this issue, Becchi and Crowley [30] introduce an improved yet simplified algorithm (we will call it BEC-CRO) which results in at most $2N$ state traversals when processing a string of length N. This work is based on the observation that all regular expression evaluations begin at a single starting state, and the vast majority of transitions among states lead back either to the starting state or its near neighbors. From this consideration and by leveraging, during automaton construction, the concept of state distance from the starting state, the algorithm achieves comparable levels of compression with respect to D^2FA, with lower provable bounds on memory bandwidth and greater simplicity. Also, the work presented in [31] focuses on the memory problem of DFAs, by proposing a technique that allows non-equivalent states to be merged, thanks to a scheme where the transitions in the DFA are labeled. In particular, the authors merge states with common destinations regardless of the characters which lead those transitions (unlike D^2FA), creating opportunities for more merging and thus achieving higher memory reduction. Moreover the authors regain the idea of bitmaps for compression purposes. Run-Length-Encoding is used in [32] to compress the transition table of DFAs. The authors show how to increase the characters processed per state traversal and present heuristics to reduce the number of memory accesses. Their work is specifically focused on an FPGA implementation. The work in [19] is based on the usual observation that DFAs are infeasible with large sets of regular expressions (especially for those which present wildcards) and that, as an alternative, NFAs alleviate the memory storage problem but lead to a potentially large memory bandwidth requirement. The reason is that multiple NFA states can be active in parallel and each input character can trigger multiple transitions. Therefore the authors propose a hybrid DFA-NFA solution bringing together the strengths of both automata: when constructing the automaton, any nodes that would contribute to state explosion retain an NFA encoding, while the others are transformed into DFA nodes. As shown by the experimental evaluation, the data structure presents a size nearly that of an NFA, but with the predictable and small memory bandwidth requirements of a DFA. Kumar et al. [33] also showed how to increase the speed of D^{2}FAs by storing more information on the edges. This appears to be a general trend in the literature even if it has been proposed in different ways: in [33] transitions carry data on the next reachable nodes, in [31] edges have different labels, and even in [18] and [20][21] transitions are no more simple pointers but a sort of "instructions". In a further comprehensive work [18], Kumar et al. analyze three main limitations of the traditional DFAs. First, DFAs do not take advantage of the fact that normal data streams rarely match more than a few initial symbols of any signature; the authors propose to split signatures such that only one portion needs to remain active, while the remaining portions can be "put to sleep" (in an external memory) under normal conditions. Second, the DFAs are extremely inefficient in following multiple partially matching signatures and this yields the so-called *state blow-up*: a new improved Finite State Machine is proposed by the authors in order to solve this problem. The idea is to construct a machine which remembers more information, such as encountering a closure, by storing them in a small and fast cache which represents a sort of history buffer. This class of machines is called History-based Finite Automaton (H-FA) and shows a space reduction close to 95%. Third, DFAs are incapable of keeping track of the occurrencies of certain sub-expressions, thus resulting in a blow-up in the number of state: the authors introduce some extensions to address this issue in the History-based counting Finite Automata (H-cFA). The idea of adding some information to keep the transition history and, consequently, reduced the number of states, has been retrieved also in [20][21], where another scheme, named extended FA (XFA), is proposed. In more details, XFA augments traditional finite automata with a finite scratch memory used to remember various types of information relevant to the progress of signature matching (e.g., counters of characters and other instructions attached to edges and states). The experimental tests performed with a large class of NIDS signatures showed time complexity similar to DFAs and space complexity similar to or better than NFAs.

2.1.2 Delta Finite Automaton

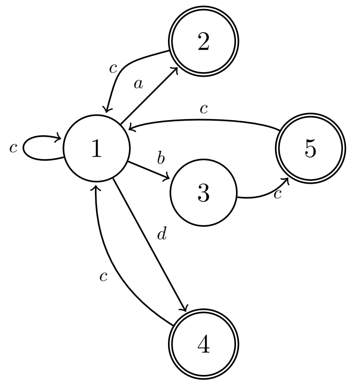
In this section we introduce δFA, a D^{2}FA-inspired automaton that preserves the advantages of D^{2}FA and requires a single memory access per input char.



(a) The DFA



(b) The D^{2}FA



(c) The δFA

Figure 2.1: Automata recognizing (a^+),(b^+c) and (c^*d^+).

2.1.2.1 Motivation through an example

In order to make clearer the rationale behind δFA construction and the differences with D^{2}FA, we start by analyzing the same example brought by Kumar et al. in [17]: the figure 2.1(a) represents a DFA on the alphabet $\{a, b, c, d\}$ that recognizes the regular expressions (a^+),(b^+c) and (c^*d^+). In figure 2.1(b) the D^{2}FA for the same set of regular expressions is shown. The main idea is to reduce the memory footprint of states by storing only a limited number of transitions for each state and a default transition to be taken for all input char for which a transition is not defined. When, for example, in figure 2.1(b) the state machine is in state 3 and the input is d, the default transition to state 1 is taken. State 1 "knows" which state to go to upon input d, therefore we jump to state 4. In this example, taking a default transition costs 1 more hop (1 more memory access) for a single input char. However, it may happen that also after taking a default transition, the destination state for the input char is not specified and another default transition must be taken, and so on. The works in [17] and [30] show how we can limit the number of hops in default paths and propose refined algorithms to define the best choice for default paths. In the example, the total number of transitions was reduced to 9 in the D^{2}FA (less than half of the equivalent DFA which has 20 edges), thus achieving a remarkable compression. However, observing the graph in fig.2.1(a), it is evident that most transitions for a given input lead to the same state, regardless of the starting state; in particular, adjacent states share the majority of the next-hop states associated with the same input chars. Then if we jump from state 1 to state 2 and we "remember" (in a local memory) the entire transition set of 1, we will already know all the transitions defined in 2 (because for each character they lead to the same set of states as 1). This means that state 2 can be described with a very small amount of bits. Instead, if we jump from state 1 to 3, and the next input char is c, the transition will not be the same as the one that c produces starting from 1; then state 3 will have to specify its transition for c. The result of what we have just described is depicted in fig.2.1(c) (except for the local transition set), which is the δFA equivalent to the DFA in fig.2.1(a). We have 8 edges in the graph (as opposed to the 20 of a full DFA) and every input char requires *a single state traversal* (unlike D^{2}FA).

2.1.2.2 Definition of δFA

As shown above, the target of δFA is to obtain a similar compression as D^{2}FA without giving up the *single state traversal per character* of DFA. The idea of δFA comes from the following observations:

- as shown in [30], most default transitions are directed to states closer to the initial state;
- a state is defined by its transition set and by a small value that represents the accepted rule (if it is an accepting state);
- in a DFA, most transitions for a given input char are directed to the same state.

By elaborating on the last observation, it becomes evident that most adjacent states share a large part of the same transitions. Therefore we can store only the differences between adjacent (or, better, "parent-child"[1]) states. This requires, however, the introduction of a supplementary structure that locally stores the transition set of the current state. The main idea is to let this local transition set evolve as a new state is reached: if there is no difference with the previous state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all cases, as a new state is read, the local transition set is updated with all the stored transitions of the state. The δFA shown in fig.2.1(c) only stores the transitions that *must* be defined for each state in the original DFA.

2.1.2.3 Construction

In alg.1 the pseudo-code for creating a δFA from a N-states DFA (for a character set of C elements) is shown. The algorithm works with the *transition table* $t[s, c]$ of the input DFA (i.e.: a $N \times C$ matrix that has a row per state and where the i-th item in a given row stores the state number to reach upon the reading of input char i). The final result is a "compressible" transition table $t_c[s, c]$ that stores, for each state, the transitions required by the δFA only. All the other cells of the $t_c[s, c]$ matrix are filled with the special LOCAL_TX symbol and can be simply eliminated by using a bitmap, as suggested in [26] and [31]. The details of our suggested implementation can be found in section 2.1.6.

Algorithm 1 Pseudo-code for the creation of the transition table t_c of a δFA from the transition table t of a DFA.

```
1:  for $c \leftarrow 1, C$ do
2:      $t_c[1, c] \leftarrow t[1, c]$
3:  end for
4:  for $s \leftarrow 2, N$ do
5:      for $c \leftarrow 1, C$ do
6:          $t_c[s, c] \leftarrow$ EMPTY
7:      end for
8:  end for
9:  for $s_{parent} \leftarrow 1, N$ do
10:     for $c \leftarrow 1, C$ do
11:         $s_{child} \leftarrow t[s_{parent}, c]$
12:         for $y \leftarrow 1, C$ do
13:             if $t[s_{parent}, y] \neq t[s_{child}, y]$ then
14:                 $t_c[s_{child}, y] \leftarrow t[s_{child}, y])$
15:             else
16:                 if $t_c[s_{child}, y] ==$ EMPTY then
17:                     $t_c[s_{child}, y] \leftarrow$ LOCAL_TX
18:                 end if
19:             end if
20:         end for
21:     end for
22: end for
```

The construction requires a step for each transition (C) of each pair of adjacent states ($N \times C$) in the input DFA, thus it costs $O(N \times C^2)$ in terms of time complexity. The space complexity is $O(N \times C)$ because the structure upon which the algorithm works is another $N \times C$ matrix. In details, the construction algorithms first initializes the t_c

[1]here the terms *parent* and *child* refer to the depth of adjacent states

matrix with EMPTY symbols and then copies the first (root) state of the original DFA in the t_c. It acts as base for subsequently storing the differences between consecutive states.
Then, the algorithm observes the states in the original DFA one at a time. It refers to the observed state as *parent*. Then it checks the *child* states (i.e.: the states reached in 1 transition from parent state). If, for an input char c, the child state stores a different transition than the one associated with any of its parent nodes, we cannot exploit the knowledge we have from the previous state and this transition must be stored in the t_c table. On the other hand, when all of the states that lead to the child state for a given character share the same transition, then we can omit to store that transition. In alg.1 this is done by using the special symbol LOCAL_TX.

Equivalent states After the construction procedure shown in alg.1, since the number of transitions per state is significantly reduced, it may happen that some of the states have the same identical transition set. If we find j identical states, we can simply store one of them, delete the other $j-1$ and substitute all the references to those with the single state we left. Notice that this operation creates again the opportunity for a new state-number reduction, because the substitution of state references makes it more probable for two or more states to share the same transition set. Hence we iterate the process until the number of duplicate states found is 0.

2.1.2.4 Lookup

Algorithm 2 Pseudo-code for the lookup in a δFA. The current state is s and the input char is c.

```
procedure Lookup(s, c)
1: read(s)
2: for $i \leftarrow 1, C$ do
3:     if $t_c[s,i] \neq$ LOCAL_TX then
4:         $t_{loc}[i] \leftarrow t_c[s,i]$
5:     end if
6: end for
7: $s_{next} \leftarrow t_{loc}[c]$
8: return $s_{next}$
```

The lookup in a δFA is computed as shown in alg.2. First, the current state must be read with its whole transition set (step 1). Then it is used to update the local transition set t_{loc}: for each transition defined in the set read from the state, we update the corresponding entry in the local storage. Finally the next state s_{next} is computed by simply observing the proper entry in the local storage t_{loc}. While the need to read the whole transition set may imply more than 1 memory access, we show in sec.2.1.5 how to solve this issue by means of a compression technique we propose. The lookup algorithm requires a maximum of C elementary operations (such as shifts and logic AND or popcounts), one for each entry to update. However, in our experiments, the number of updates per state is around 10. Even if the actual processing delay strictly depends on many factors (such as clock speed and instruction set), in most cases, the computational delay is negligible with respect to the memory access latency.

In fig.2.2 we show the transitions taken by the δFA in fig.2.1(c) on the input string *abc*: a circle represents a state and its internals include a bitmap (as in [26] to indicate which transitions are specified) and the transition set. The bitmap and the transition set have been defined during construction. It is worth noticing that the "duplicate" definition of transitions for character *c*. We have to specify the *c*-transition for state 2 even if it is the same as the one defined in state 1, because state 2 can be reached also from state 3 which has a different next state for *c*. We start ($t = 0$) in state 1 that has a fully-specified transition set. This is copied into the local transition set (below). Then we read the input char *a* and move ($t = 1$) to state 2 that specifies a single transition toward state 1 on input char *c*. This is also an accepting state (underlined in figure). Then we read *b* and move to state 3. Note that the transition to be taken now is not specified within state 2 but it is in our local transition set. Again state 3 has a single transition specified, that this time changes the corresponding one in the local transition set. As we read *c* we move to state 5 which is again accepting.

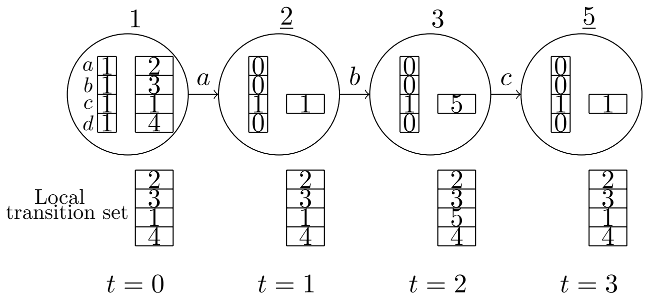
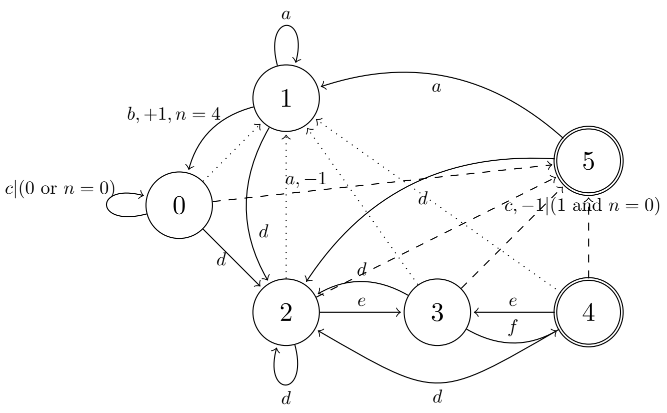


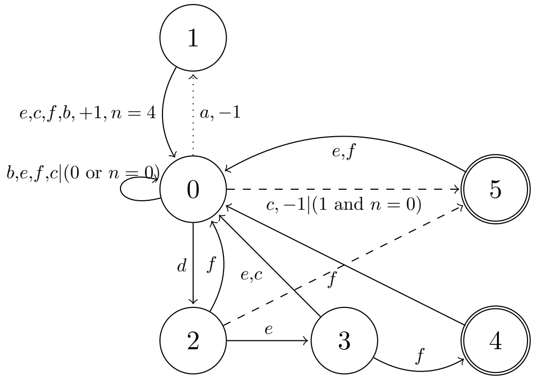
Figure 2.2: δFA internals: a lookup example.

2.1.3 Application to H-cFA and XFA

One of the main advantage of our δFA is that it is orthogonal to many other schemes. Indeed, very recently, two major DFA compressed techniques have been proposed, namely H-cFA [18] and XFA [20][21]. Both these schemes address, in a very similar way, the issue of state blow-up in DFA for multiple regular expressions, thus candidating to be adopted in platforms which provide a limited amount of memory, as network processors, FPGAs or ASICs. The idea behind XFAs and H-cFA is to trace the traversal of some certain states that corresponds to closures by means of a small scratch-memory. Normally those states would lead to state blow-up; in XFAs and H-cFA flags and counters are shown to significantly reduce the number of states.
The application of δFA to H-cFA and XFA (which is tested in sec.2.1.7) is obtained by storing the "instructions" specified in the edge labels only once per state. Moreover edges are considered different also when their specified "instructions" are different. To better clarify the idea, an example of the application to H-cFA (again taken from a previous paper [18]) is reported in fig.2.3(a). The aim is to recognize the regular expressions *.*ab[^a]*c* and *.*def*, and labels include also conditions and operations that operate on a flag (set/reset with +/-1) and a counter *n* (for more details refer to [18]). A DFA would need 20 states and a total of 120 transitions, the corresponding



(a) The H-cFA. Dashed and dotted edges have same labels, respectively $c, -1|(1 \text{ and } n = 0)$ and $a, -1$. Not all edges are shown to keep the figure readable. The real number of transitions is 38.



(b) The δH-cFA. Here all the 18 transitions are shown.

Figure 2.3: Automata recognizing `.*ab[^a]*c` and `.*def`

H-cFA (fig.2.3(a)) uses 6 states and 38 transitions, while the δFA representation of the H-cFA (fig.2.3(b)) requires only 18 transitions.

2.1.4 Compressing char-state pairs

In a δFA, the size of each state is not fixed because an arbitrary number of transitions can be present, and therefore state pointers are required, which generally are standard memory addresses. They constitute a significant part of the memory occupation associated with the DFA data structure, so we propose here a compression technique which remarkably reduces the number of bits required for each pointer. Such an algorithm is fully compatible with δFA and most of the other solutions for DFA compression already shown in section 2.1.1. Our algorithm (hereafter referred to as *char-state compression* or simply *C-S*) is based on a heuristic which is verified by several standard rule sets: in most cases, the edges reaching a given state are labelled with the same character. Table 2.1 shows, for different available data sets (see section 2.1.7 for more details on sets) the percentage of nodes which are reached only by transitions corresponding to a single character over the total number of nodes.

Data set	p_{1char} (%)	r_{comp} (%)	η_{acc}	T_S (KB)
Snort34	96	59	1.52	27
Cisco30	89	67	1.62	7
Cisco50	83	61	1.52	13
Cisco100	78	59	1.58	36
Bro217	96	80	1.13	11

Table 2.1: Percentage of states reached by edges with the same one label (p_{1char}), *C-S* compression (r_{comp}), average number of scratchpad accesses per lookup (η_{acc}) and indirection-table size (T_S).

As a consequence, a consistent number of states in the DFA can be associated with a single character and can be referred to by using a “relative” address. More precisely, all the states reached by a transition labelled with character c will be given a “relative” identifier (hereafter simply *relative-id*); since the number of such states will be smaller than the number of total states, a relative-id will require a lower number of bits than an absolute address. In addition, as the next state is selected on the basis of the next input char, only its relative-id has to be included in the state transition set, thus requiring less memory space.
In a D^{2}FA, where a default transition accounts for several characters, we can simply store it as a relative-id with respect to the first character associated with it. The absolute address of the next state will be retrieved by using a small indirection table, which, as far as our experimental results show, will be small enough to be kept in local (or in a scratchpad) memory, thus allowing for fast lookup. It is clear that such a table will suffer from a certain degree of redundancy: some states will be associated with several relative-ids and their absolute address will be reported more than once. In the next subsection we then propose a method to cope with such a redundancy, in the case it leads to an excessive memory occupation.

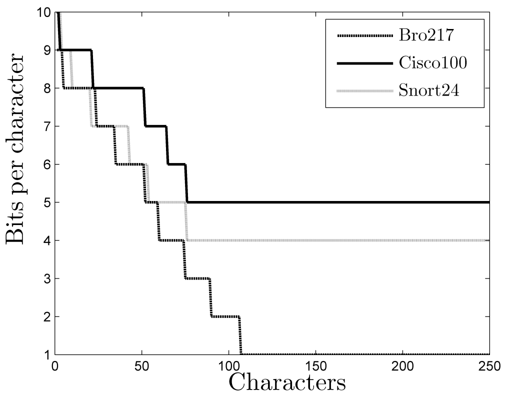


Figure 2.4: Distribution of the number of bits used for a relative identifier with our compression scheme for standard rule sets.

Figure 2.4 shows the distribution of the number of bits that may be used for a relative-id when applying our compression scheme to standard rule sets. As it can be noticed, next state pointers are represented in most cases with very few bits (less than five); even in the worst case, the number of bits is always below ten. In the second column of table 2.1, we show the compression rate achieved by *C-S* with respect to a naive implementation of DFA for the available data sets. As it appears from the table, the average compression is between 60% and 80%.

2.1.4.1 Indirection Table Compression

As claimed above, the implementation of *Char-State compression* requires a lookup in an indirection table which should be small enough to be kept in local memory. If several states with multiple relative-ids are present in such a table, this might be an issue. For this reason we present a lookup scheme which offers an adaptive trade-off between the average number of memory accesses and the overall memory occupation of the table.
The table that we use in our scheme encompasses two kinds of pointers: absolute pointers and local ones. When a state has a unique relative-id, its absolute address is written in the table; otherwise, if it has multiple relative-ids, for each one of them the table reports a pointer to a list of absolute addresses; such a pointer will require a consistently smaller number of bytes than the address itself. An absolute address is then never repeated in the table, thus preventing from excessive memory occupation. Such a scheme is somewhat self-adapting since, if few states have multiple identifiers, most of the translations will require only one memory access, while, if a consistent amount of redundancy is present, the translation will likely require a double indirection, but the memory occupation will be consistently reduced. Notice that the presence of different length elements in the table poses no severe issues: since the relative address is arbitrary, it is sufficient to assign lower addresses to nodes which are accessible with only one lookup and higher addresses to nodes requiring double indirection, and to keep a threshold value in the local memory. The results in terms

of memory accesses and size of such a scheme applied to the available data sets are reported in tab.2.1.

2.1.5 δFA with C-S

The *C-S* can be easily integrated within the δFA scheme and both algorithms can be cross-optimized. Indeed, *C-S* helps δFA by reducing the state size thus allowing the read of a whole transition set in a single memory access on average. On the other hand, *C-S* can take advantage of the same heuristic of δFA: successive states often present the same set of transitions. As a consequence, it is possible to parallelize the retrieval of the data structure corresponding to the next state and the translation of the relative address of the corresponding next-state in a sort of "speculative" approach. More precisely, let s and $s+1$ be two consecutive states and let us define A_s^c as the relative address of the next hop of the transition departing from state s and associated with the character c. According to the previously mentioned heuristic it is likely that $A_s^c = A_{s+1}^c$; since, according to our experimental data (see sec.2.1.7), 90% of the transitions do not change between two consecutive states, we can consider such an assumption to be verified with a probability of roughly 0.9. As a consequence, when character c is processed, it is possible to parallelize two memory accesses:

- retrieve the data structure corresponding to state $s+1$;
- retrieve the absolute address corresponding to A_{s+1}^c in the local indirection table.

In order to roughly evaluate the efficiency of our implementation in terms of the state lookup time, we refer to a common underlying hardware architecture (described in section 2.1.6). It is pretty common [34] that the access to a local memory block to be than twice as faster than that of to an off-chip memory bank: as a consequence, even if a double indirection is required, the address translation will be ready when the data associated with the next state will be available. If, as it is likely, $A_s^c = A_{s+1}^c$, it will be possible to directly access the next state (say $s+2$) through the absolute pointer that has just been retrieved. Otherwise, a further lookup to the local indirection table will be necessary.

Dataset	# of regex	ASCII length range	% Regex w/ wildcards (*,+,?)	Original DFA	
				# of states	# of transitions
Snort24	24	6-70	83.33	13886	3554816
Cisco30	30	4-37	10	1574	402944
Cisco50	50	2-60	10	2828	723968
Cisco100	100	2-60	7	11040	2826240
Bro217	217	5-76	3.08	6533	1672448

Table 2.2: Characteristics of the rule sets used for evaluation.

Such a parallelization can remarkably reduce the mean time needed to examine a new character. As an approximate estimation of the performance improvement, let us suppose that our assumption (i.e. $A_s^c = A_{s+1}^c$) is verified with probability $p = 0.9$, that one access to on-chip memory takes $t_{on} = 4T$ and to an external memory $t_{off} = 10T$ [34], and that an address translations requires $n_{trans} = 1.5$ memory accesses (which is reasonable according to the fourth column of table 2.1). The mean delay will be then:

$$\overline{t_{par}} = (1-p)(t_{off} + n_{trans} \times t_{on}) + p \times t_{off} = 10.6T$$

This means that even with respect to the implementation of δFA the *C-S* scheme increases the lookup time by a limited 6%. On the contrary, the execution of the two tasks serially would required:

$$\overline{t_{ser}} = (t_{off} + n_{trans} \times t_{on}) = 16T$$

The parallelization of tasks results then in a rough 50% speed up gain.

2.1.6 Implementation

The implementation of δFA and *C-S* should be adapted to the particular architecture of the hardware platform. However, some general guidelines for an optimal deployment can be outlined. In the following we will make some general assumptions on the system architecture; such assumptions are satisfied by many network processing devices (e.g. the Intel IXP Network Processors [35]). In particular, we assume our system to be composed by:

- a standard 32 bit processor provided with a fairly small local memory (let us suppose a few KBs); we consider the access time to such a memory block to be of the same order of the execution time of an assembly level instruction (less than ten clock cycles);
- an on-chip fast access memory block (which we will refer to as scratchpad) with higher storage capacity (in the order of 100 KB) and with an access time of a few dozens of clock cycles;
- an off-chip large memory bank (which we will refer to as external memory) with a storage capacity of dozens of MBs and with an access time in the order of hundreds of clock cycles.

We consider both δFA and *Char-State compression* algorithms. As for the former, two main kinds of data structures are needed: a unique local transition set and a set of data structures representing each state (kept in the external memory). The local transition set is an array of 256 pointers (one per character) which refer to the external memory location of the data structure associated with the next state for that input char; since, as reported in table 2.3(b) , the memory occupation of a δFA is generally smaller than 1 MB, it is possible to use a 20 bit-long offset with respect to a given memory address instead of an actual pointer, thus achieving a consistent compression. A δFA state is, on the contrary, stored as a variable-length structure. In its most general form, it is composed by a 256 bit-long bitmap (specifying which valid transition are already stored in the local transition set and which ones are instead stored within the state) and a list of the pointers for the specified transitions, which, again, can be considered as 20 bit offset values. If the number of specified transitions within a state is small enough, the use of a fixed size bitmap is not optimal: in these cases, it is possible to use a more compact structure, composed by a plain list of character-pointer couples. Note that this solution allows for memory saving when less than 32 transitions have to be updated in the local table. Since in a state data structure a pointer is associated with a unique character, in order to integrate *Char-State compression* in this scheme it is sufficient to substitute each absolute pointer with a relative-id. The only additional structure consists of a character-length correspondence list, where the length of the relative-ids associated with each character is stored; such an information is necessary to parse the pointer lists in the node and in the local transition set. However, since the maximum length for the identifiers is generally lower than 16 bits (as it is evident from figure 2.4), 4 bits for each character are sufficient. The memory footprint of the character-length table is well compensated by the corresponding compression of the local transition set, composed by short relative identifiers (our experimental results show a compression of more than 50%). Furthermore, if a double indirection scheme for the translation of relative-ids is adopted, a table indicating the number of unique identifiers for each character (the threshold value we mentioned in section 2.1.4.1) will be necessary, in order to parse the indirection table. This last table (that will be at most as big as the compressed local transition table) can be kept in local memory, thus not affecting the performance of the algorithm.

(a) Transitions reduction (%). For δFA also the percentage of duplicate states is reported.

Dataset	D^2FA					BEC-CRO	δFA	
	$DB = \infty$	$DB = 14$	$DB = 10$	$DB = 6$	$DB = 2$		trans.	dup.states
Snort24	98.92	98.92	98.91	98.48	89.59	98.71	96.33	0
Cisco30	98.84	98.84	98.83	97.81	79.35	98.79	90.84	7.12
Cisco50	98.76	98.76	98.76	97.39	76.26	98.67	84.11	1.1
Cisco100	99.11	99.11	98.93	97.67	74.65	98.96	85.66	11.75
Bro217	99.41	99.40	99.07	97.90	76.49	99.33	93.82	11.99

(b) Memory compression (%).

Dataset	D^2FA					BEC-CRO	δFA + C-S
	$DB = \infty$	$DB = 14$	$DB = 10$	$DB = 6$	$DB = 2$		
Snort24	95.97	95.97	95.94	94.70	67.17	95.36	95.02
Cisco30	97.20	97.20	97.18	95.21	55.50	97.11	91.07
Cisco50	97.18	97.18	97.18	94.23	51.06	97.01	87.23
Cisco100	97.93	97.93	97.63	95.46	51.38	97.58	89.05
Bro217	98.37	98.34	95.88	95.69	53	98.23	92.79

Table 2.3: Compression of the different algorithms in terms of transitions and memory.

2.1.7 Experimental Results

This subsection shows a performance comparison among our algorithm and the original DFA, D^2FA and BEC-CRO. The experimental evaluation has been performed on some data sets of the Snort and Bro intrusion detection systems and Cisco security appliances [15]. In details, such data sets, presenting up to hundreds of regular expressions, have been randomly reduced in order to obtain a reasonable amount of memory for DFAs and to observe different statistical properties. Such characteristics are summarized in table 2.2, where we list, for each data set, the number of rules, the ascii length range and the percentage of rules including "wildcards symbols" (i.e. *, +, ?). Moreover, the table shows the number of states and transitions and the

amount of memory for a standard DFA which recognizes such data sets, as well as the percentage of duplicated states. The choice of such data sets aims to mimic the size (in terms of DFA states and regular expressions) of other sets used in literature [31][17][19],[30] in order to obtain fair comparisons.
Tables 2.3 illustrate the memory compression achieved by the different algorithms. We have implemented the code for our algorithm, while the code for D^2FA and BEC-CRO is the *regex-tool* [36] from Michela Becchi (for the D^2FA the code runs with different values of the diameter bound, namely the diameter of the equivalent maximum weight spanning tree found in the space reduction graph [17]; this parameter affects the structure size and the average number of state-traversals per character). By means of these tools, we build a standard DFA and then reduce states and transitions through the different algorithms. The compression in tab. 2.3(a) is simply expressed as the ratio between the number of deleted transitions and the original ones (previously reported in tab.2.2) , while in 2.3(b) it is expressed considering the overall memory saving, therefore taking into account the different state sizes and the additional structures as well. Note also, in the last column of tab.2.3(a) , the limited but effective state-reduction due to the increased similarity of states obtained by the δFA (as described in sec.2.1.2.3). Although the main purpose of our work is to reduce the time complexity of regular expression matching, our algorithm achieves also a degree of compression comparable to that of D^2FA and BEC-CRO, as shown by tab.2.3. Moreover, we remark that our solution is orthogonal to these algorithms (see sec.2.1.3), thus allowing further reduction by combining them.

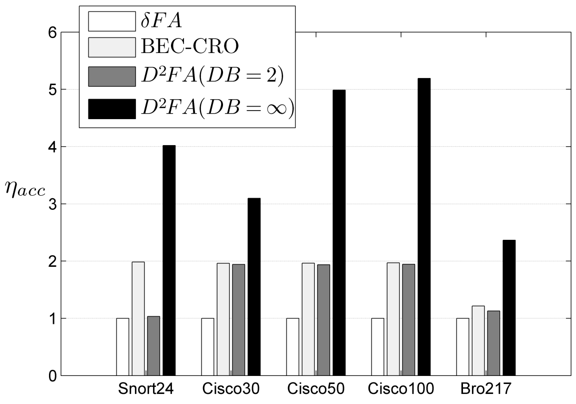


Figure 2.5: Mean number of memory accesses for δFA, BEC-CRO and D^2FA for different datasets.

Figure 2.5 shows the average number of memory accesses (η_{acc}) required to perform pattern matching through the compared algorithms. It is worth noticing that, while the integration of *C-S* into δFA (as described in sec.2.1.5) reduces the average state size, thus allowing for reading a whole state in slightly more than $1 (< 1.05)$ memory accesses, the other algorithms require more accesses, thus increasing the lookup time. We point out that the mean number of accesses for the integration of δFA and *C-S* is not included in the graph in that *C-S* requires accesses to a local scratchpad memory, while the accesses the figure refers to are generally directed to an external, slower memory block; therefore it is difficult to quantify the additional delay introduced by *C-S*. However, as already underlined in section 2.1.5, if an appropriate parallelization scheme is adopted, the mean delay contribution of *C-S* can be considered nearly negligible on most architectures.

Dataset	# of states	# of trans. XFA	# of trans. δXFA	Compr. %
c2663-2	14	3584	318	92
s2442-6	12	3061	345	74.5
s820-10	23	5888	344	94.88
s9620-1	19	4869	366	92.70

Table 2.4: Number of transitions and memory compression by applying δFA+*C-S* to XFA.

Finally, table 2.4 reports the results we obtained by applying δFA and *C-S* to one of the most promising approach for regular expression matching: XFAs [20][21] (thus obtaining a δXFA). The data set (courtesy of Randy Smith) is composed of single regular expressions with a number of closures that would lead to a state blow-up. The XFA representation limits the number of states (as shown in the table).

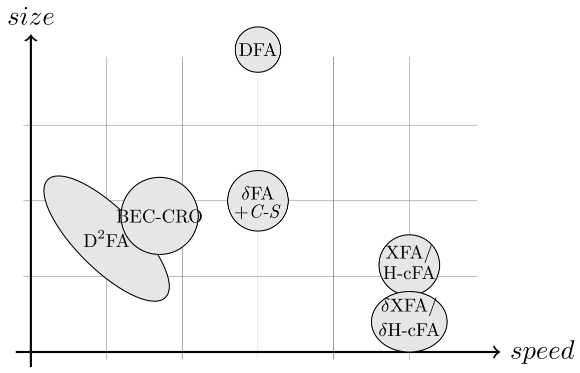


Figure 2.6: Comparison of speed performance and space requirements for the different algorithms.

By adopting δFA and *C-S* we can also reduce the number of transitions with respect to XFAs and hence achieve a further size reduction. In details, the reduction achieved is more than 90% (except for a single case) in terms of number of transitions, that corresponds to a rough 90% memory compression (last column in the table). The memory requirements, both for XFAs and δXFAs, are obtained by storing the "instructions" specified in the edge labels only once per state. Figure 2.6 resumes all the evaluations by mixing speed performance (in terms of memory accesses) and space requirements in a qualitative graph (proportions are not to be considered real). It is evident that our solution almost achieves the compression of D^2FA and BEC-CRO, while it proves higher speed (as that of DFA). Moreover, by combining our scheme with other ones, a general performance increase is obtained, as shown by the integration with XFA or H-cFA.

2.2 Second order delta enconding to improve DFA efficiency

In the previous section, we have introduced a compact representation scheme (named δFA) which is based on the observation that, since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only. This requires, however, the introduction of a supplementary structure that locally stores the transition set of the current state. The main idea is to let this local transition set evolve as a new state is reached: if there is no difference with the previous state in terms of the next state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. This idea was inspired by D^{2}FA [17], which introduces default transitions (and a "path delay") for reducing transitions, but, unlike the previous algorithms, δFA examines one state per character only, thus reducing the number of memory accesses and speeding up the overall lookup process. In this work, we present a novel automaton which takes advantage of the ideas of δFA and adds the concept of "temporary transition". It extends the δFA main assumption some step further: while δFA specifies the transition set of a state with respect to its direct parents, the adoption of 2-step "ancestors" (in this definition a direct parent is a 1-step ancestor) increases the chances of compression. As we will show in the following, the best approach to exploit this second order dependence is to define the transitions of the states between the ancestors and the child as "temporary". This, however, introduces a new problem during the construction process: the optimal construction (in terms of memory or transition reduction) appears to be an NP-complete problem. Therefore, a direct and oblivious approach is chosen for simplicity. Results (on real rule-sets from Snort, Bro and Cisco devices) show that our simple approach do not differ significantly from the optimal (if ever reachable) construction. Since the technique we propose is an extension to δFA that exploits second order dependence, we name this scheme δ^2FA.

2.2.1 The Main idea of δ^2FA

Consider again the DFA in 2.7(a). Although the δFA in fig. 2.7(b) shows a remarkable saving in terms of transitions with respect to the standard DFA, its main assumption (all parents must share the same transition for a given character) somewhat limits the effectiveness of the compression. In the example, all the transitions for character c are specified (and hence stored) for all the 5 states, because of a single state 3 that defines a different transition (the transition for c is directed to state 1 for states $1, 2, 4$ and 5, while 3 defines an edge to 5). Notice that this is due to the strict definition of δFA rules that do not "see" further than a single-hop: the transition set of a state is stored as the difference with respect to all its direct parents.
Intuitively, just as a D^{2}FA with long default-transitions paths compresses better than

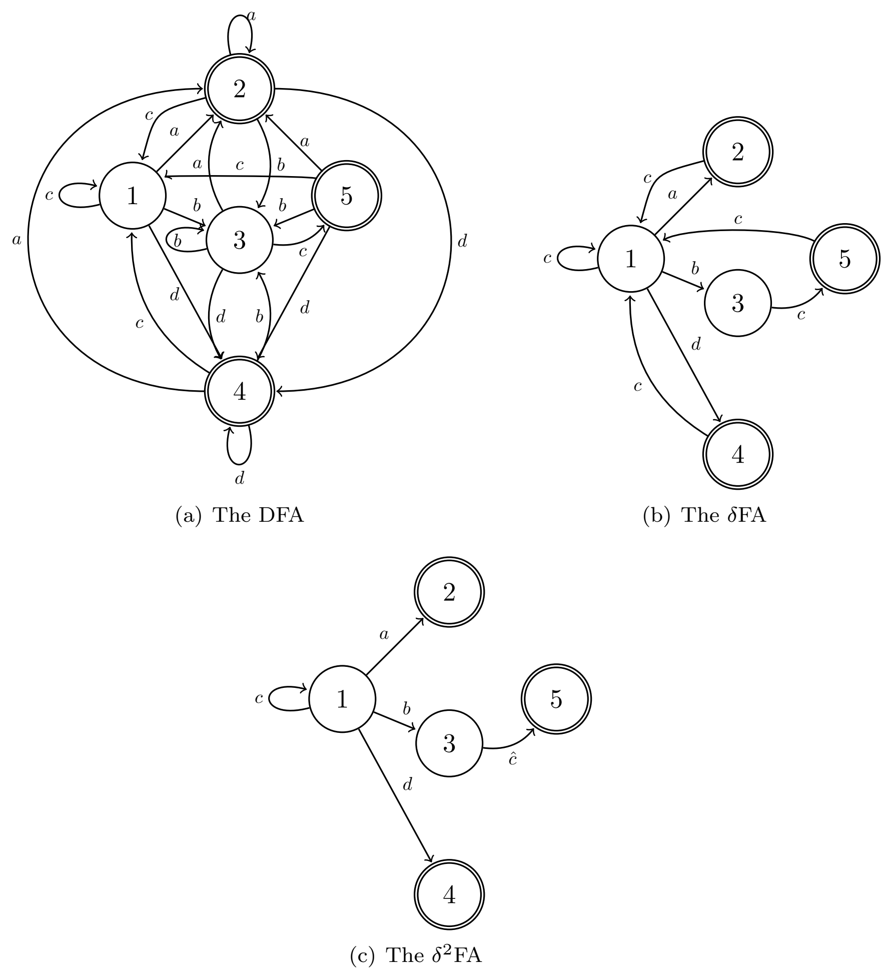


Figure 2.7: Automata recognizing (a^+), (b^+c) and (c^*d^+).

a bounded D^{2}FA with B=2 [17], by relaxing the definition of "parents" to "grandparents" (i.e., 2-step neighbor nodes) the effectiveness of the δFA approach increases because of the larger number of possibilities.
However, a blind adoption of this concept does not provide better results in δFA: for instance, in fig. 2.7(b) defining the transitions for c as difference with respect to all the "grandparents" still would not allow to eliminate any new transition. Moreover this scheme would require to store 2 local transition sets (doubling the local memory needed).
A better approach is, instead, to define the transition for c in state 3 as "temporary", in the sense that it does not get stored in the local transition set. In this way, we force the transition to be defined uniquely within state 3 and not to affect its children. This means that, whenever we read state 3, the transition for c in the local transition set is not updated, but it remains as it was in its parents. Then, we can avoid storing the transitions for c in states $2, 4$ and 5, as shown in fig. 2.7(c) where the temporary transition is signaled with $\hat{c}$.
By defining temporary transitions, we effectively exploit 2-nd order relationships among states in a simple way, without incurring in the need for 2-times larger local memories.

2.2.1.1 Lookup

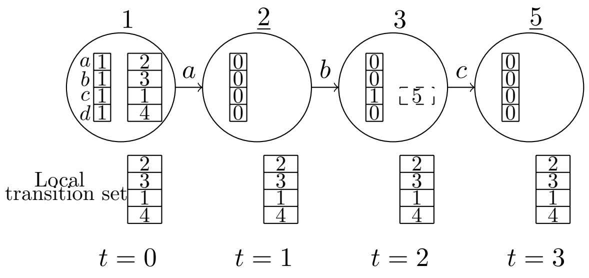


Figure 2.8: δ^2FA internals: a lookup example.

The lookup in a δ^2FA differs very slightly from that of δFA. The only difference concerns the way we handle temporary transitions: temporary transitions are valid within their state but they are not stored in the local transition set. Fig. 2.8 shows also an example of the lookup process for a δ^2FA: the whole transition set of state 1 (where we start at time $t = 0$) is copied into the local transition set. Then by char a, we move ($t = 1$) to state 2 which does not specify any transition. When we read b ($t = 2$), we move to state 3, where a temporary transition (dashed box) is specified: this transition is valid only within state 3. Finally ($t = 3$) we read c, take the temporary transition, and end up in state 5.

2.2.1.2 Construction

The construction process of the δ^2FA requires the corresponding δFA to be constructed beforehand and used as input. Then, the process works by recognizing subsets of

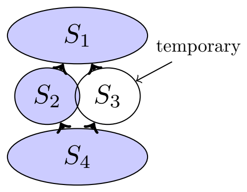


Figure 2.9: Schematic view of the problem. Same color means same properties. If the properties of S_3 are set temporary, the ones in S_1 can be avoided.

nodes where a transition for a given character can be defined as temporary. In fig. 2.9, nodes are shown as divided into sets according to their parent-child relationships (highlighted by the bold arrows) and their transitions (for a given character). In particular, all nodes with the same transition for a given char x share the same color: sets S_1, S_2 and S_4 all provide the same transition for char x, while S_3 defines a different next state for x. If we set all the transitions for x in S_3 as temporary, we can avoid storing the transition for x in S_1.
In a real implementation, in order to recognize the nodes where a transition for a given character can be defined as temporary, for each char x of each state s, if the corresponding transition $t[s,x]$ in the δFA is stored (i.e., it is different from that $t[p,x]$ of all its parents) the following steps are required:

- a search is performed in all the children of s: whenever at least a child has the same transition $t[p,x]$ of its "grandparents", the second step follows;
- check all the other parents (except for s) of such a subset of children in order to check if they have the same transition $t[p,x]$;
- in this case, the transition $t[s,x]$ in s can be set as temporary and the process ends.

The process is also described in alg. 3 where, for the sake of readability, we adopt the same notation of fig. 2.9.
A few remarks (which ultimately result in constraints in the construction process) can be explained by referring to fig. 2.9 (where the transitions for x in S_3 are set temporary):

1. no state in S_4 can have a temporary transition for x. The reason is simple: a temporary transition for x in the parents S_4 means that such a transition does not modify the local transition table and therefore we have no way to "remember" the next-state when (after some hops) we reach the children S_1;
2. all children states in S_1 must have specified transitions for x, because if the transitions in S_3 are temporary and an un-specified transition exists in a state $s_j \in S_1$, the ultimate result is that $t[s_j,x] = t[S_4,x]$ while s_j was meant to inherit $t[S_3,x]$.

Hence, this process introduces some constraints and, as usual when dealing with constraints on graphs, this creates new problems: as described above, when setting

Algorithm 3 Pseudo-code for the creation of the transition table t_2 of a δ^2FA from the transition table t of a δFA.

```
1: $t_2 \leftarrow t$
2: for all state $s$ in $\delta$FA do
3:     for all char $c$ do
4:         if $t[s,c] \neq$ LOCAL_TX then
5:             $S_4 \leftarrow \{$ parents of $s\}$
6:             if $t[s_j,c] \quad \forall s_j \in S_4$ are equal and specified then
7:                 $S_1 \leftarrow \{$ children of $s\}$
8:                 if $\exists s_j \in S_1$ s.t $t[s_j,c] ==$ LOCAL_TX then
9:                     break
10:                end if
11:                if $\exists s_j \in S_1$ s.t $t[s_j,c] == t[S_4,c]$ then
12:                    $S_2 \leftarrow \{$ parents of $s_j\} \setminus s$
13:                    if $t[S_2,c] == t[s_j,c] == t[S_4,c] \neq t[s,c]$ then
14:                        $t_2[s,c] \leftarrow TEMP\_TX$
15:                        delete $t_2[s_j,c]$
16:                    end if
17:                end if
18:            end if
19:        end if
20:    end for
21: end for
```

a subset y of transitions as temporary, we must rely on some other transitions (the granparents of y) to be non-temporary. This can be classified as a graph-coloring problem which is known to be NP-hard.
Because of this severe problem, we adopt a straight and oblivious construction: we construct the δ^2FA in a single run by observing all the transitions and setting all the transitions that satisfy the above-mentioned constraints as temporary. This solution is very fast because it does not explore the whole solution domain and simply gives up the idea of optimality. While this may appear unusual and is certainly non-optimal, it is however motivated by a number of experimental results (reported in the following section), where this approach does not differ significantly from the optimal setting (if ever reachable) in terms of transitions reduction. Moreover, notice that the optimal construction would require an exhaustive search of all the solution domain, thus questioning the advantages of the optimal setting.

2.2.2 Experimental Results

In this subsection we report the experimental results of our proposed technique (δ^2FA) applied to real-world regular expression-set from IDS/IPSs such as Snort and BRO and from Cisco security devices [15]. As a first set of results, in order to motivate the simplistic approach to the construction of δ^2FA, we compare the best (if ever reachable) construction and the simple approach we adopt. Since the ultimate goal of this work is to come up with an efficient way to further reduce the number of transitions to store in a δFA, the comparison is expressed in terms of deleted transitions. The results in tab. 2.5 show the ratio between the number of deleted (and temporary) transitions of our simple approach and the maximum number of deleted (and temporary) transitions we may have in the optimal setting. The latter is computed by accepting the violation of the two constraints described in the previous section.

Dataset	Cisco30	Cisco50	Snort24	Snort31	Bro217
Del. ratio	97%	89%	100%	99%	99%
Temp. ratio	84%	76%	100%	98%	99%

Table 2.5: Simple vs. Optimal approach: ratio of deleted and temporary transitions.

Hence, in this sense, this optimal value is actually a bound. The values in the table suggest the simple approach is effective and provides very good results, reaching the maximum number of deleted transitions in almost all the cases.

(a) Transitions reduction (%).

Dataset	D^2FA		BEC-CRO	δFA	δ^2FA
	$DB = \infty$	$DB = 2$			
Snort24	98.92	89.59	98.71	96.33	96.82
Cisco30	98.84	79.35	98.79	90.84	92.01
Cisco50	98.76	76.26	98.67	84.11	86.11
Cisco100	99.11	74.65	98.96	85.66	86.90
Bro217	99.41	76.49	99.33	93.82	94.30

(b) Memory compression (%).

Dataset	D^2FA		BEC-CRO	δFA	δ^2FA
	$DB = \infty$	$DB = 2$			
Snort24	95.97	67.17	95.36	95.02	95.90
Cisco30	97.20	55.50	97.11	91.07	92.65
Cisco50	97.18	51.06	97.01	87.23	89.03
Cisco100	97.93	51.38	97.58	89.05	90.3
Bro217	98.37	53	98.23	92.79	93.4

Table 2.6: Compression of the different algorithms. In (b) the results for δFA and δ^2FA include char-state compression.

Tab. 2.6 shows a performance comparison among δFA and δ^2FA (which include also the *Char-State* encoding scheme for further memory compression, as explained in 2.1.4) and the most efficient previous solutions. For D^2FA and BEC-CRO, we use the code of *regex-tool* [36], which builds a standard DFA and then reduces states and transitions through the different algorithms. In particular, for the D^2FA the code runs with two different values of the bound B (i.e., 2 and ∞), which is a parameter that affects the structure size and the average number of state-traversals per character [17]. The compression in tab. II(a) is simply expressed as the ratio between the number of deleted transitions and the original ones, while in tab. II(b) it is expressed by considering the overall memory consumption, therefore taking into account the different state sizes and the additional structures as well. Our algorithms achieve a degree of compression comparable to that of D^2FA and BEC-CRO, while allowing for a higher lookup speed by preserving one transition per character. This is the main strength of our scheme, which allows for reducing lookup time by exploiting the

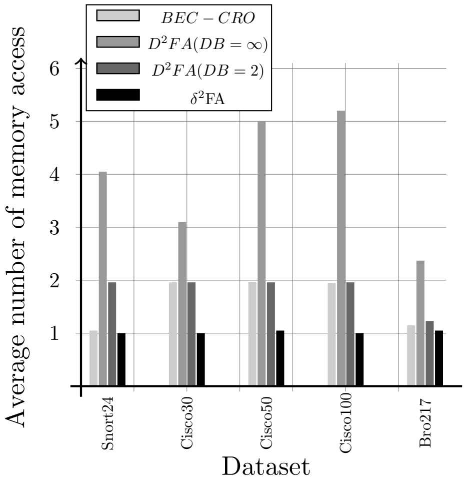


Figure 2.10: Mean number of memory accesses.

adoption of wide memory accesses which are very common in DRAMs. As shown by results, δ^2FA provides an improvement with respect to δFA at practically no cose, since it requires a minimal change in the lookup algorithm. Finally since our solutions are orthogonal to previous algorithms, a further reduction is possible by combining them. Fig. 2.10 shows the average number of memory accesses required to perform pattern matching through the compared algorithms. It is worth noticing that, while δ^2FA (just as δFA) needs about < 1.05 accesses (more than 1 because of the integration with the *Char-State* scheme), the other algorithms require more accesses, thus increasing the lookup time.

2.3 Homomorphic encoding of DFAs

In this section we propose a solution to increase the speed of regular expression searching techniques by multiplying the amount of bytes processed per cycle while also reducing memory requirements. Indeed, very few works have explored this possibility. The main reason is that, when processing k bytes per step, 256^k transitions per state are needed, so even observing only 2 bytes per cycle would require each DFA state to define 65536 transitions. Of course, the amount of states reachable in one-hop from a given state is not that large, on the contrary it is limited and concentrated on its average. Such fact is exploited in this work in order to define a simple and effective way to build small-sized and fast DFAs that process k bytes per step. This involves the definition and application of an homomorphism [37], hence we name our DFA representation Homomorphic-DFA (h-DFA).

2.3.1 Related works

The current trend in research and industry is to use DFAs to represent regular expressions, in order to obtain higher performance, while trying to solve their problems in terms of memory requirements. Many recent works have been presented with the aim of reducing their memory footprint. For a complete survey of these works, please refer to section 2.1.1. This work focuses on speed as main issue for current regular expression searching techniques. Very few works have explored the possibility to increase searching–speed in DFAs. Basically, the idea of these previous works is to multiply the amount of bytes processed per cycle, thus working with 2, 3 or 4-byte strides. However, even observing only 2 bytes per cycle would require each DFA state to include 2^{16} transitions. To solve this problem, the authors of [32] suggest a solution by observing that in actual FAs the number of different transitions (even when k bytes are processed) is more limited. In particular, they propose the use of Equivalent Character Identifiers defining the set of input words (strides of k bytes) which produce transitions to the same next state. Moreover, Run Length Encoding is used to encode the transition table. Such an approach is not general and presents some limitations, as highlighted by [38]. Indeed, it is not feasible in contexts where big DFAs (more than 100 states) and/or large compressed alphabets are involved. Therefore, the authors of [38] try to make a k-DFA feasible by taking advantage of alphabet-reduction and default transition compression. The use of alphabet-reduction, as well as in [32], is justified by the fact that, when the number of processed bytes increases, the automaton actually uses only a small subset of the entire alphabet. Instead, the default transition compression acts by removing the transitions redundancy present in a DFA. Indeed, if the stride doubles, the number of transitions in the DFA increases quadratically, but the number of states does not; therefore, intuitively, the fraction of distinct transitions decreases and the transition redundancy tends to increase.

2.3.2 An efficient representation for DFAs

In the following we introduce the basics of our scheme. We want to succinctly describe the outgoing transitions for each state, so that, when computing the corresponding k-step DFA, we have to combine a small amount of one-step transitions.
Our main idea is to group all the symbols that produce a transition to a given node into a subset and find a series of functions that, only when applied to such a subset, provides a specific result or a set of results. When applied to all the other symbols, the result must be different. More formally, in each state, for each subset of symbols S_j that produces a transition to a node n_j, we look for a function $h_j(c)$ such that

$$h_j(c) = x_j \in \begin{cases} X_j & \forall c \in S_j \\ U \setminus X_j & \forall c \notin S_j \end{cases} \tag{2.1}$$

where U is the image of $h_j(c)$ and X_j is the subset of the image of $h_j(c)$ for $c \in S_j$.
By means of this series of functions h_j, we can describe the transition set of each node as an array of tuples:

$$(h_1 : x_{1,1}, \dots, x_{1,N_1} : n_1) \dots (h_d : x_{d,1}, \dots, x_{d,N_d} : n_d) \tag{2.2}$$

where d is the state outdegree, $n_1 \dots n_d$ are the reachable states, $x_{k,1} \dots x_{k,N_k}$ are the different values that h_k takes in S_k or, in other words, a representation of X_k and N_k is the cardinality of X_k.
Such a representation helps reducing the redundancy of DFAs as regular Alphabet

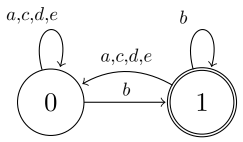


Figure 2.11: A very simple DFA

Compression Tables: it requires to store $\sum_k N_k$ values, d functions and d pointers to next states. As an example of the compactness of such representation, let us observe the transition set of the DFA state 1 in fig.2.11. In this case, the characters a,c,d and e all belong to a subset S_0 that produces a transition to state 0. Therefore we can describe the transition set with two tuples only:

$$\{h_0{:}X_0{:}0\}, \quad \{h_1{:}X_1{:}1\}$$

where h_0 and h_1 are defined as in (2.1): when h_0 is applied to a,c,d and e, the result is in X_0, while $h_1(b) \in X_1$.
By defining a set of functions to a DFA, we exploit the properties of inverse homomorphisms applied to DFAs. An homomorphism [37] is an application that maps symbols to strings belonging to a language L. An inverse homomorphisms translate strings of a language L into symbols belonging to a given alphabet. From our point of view, by grouping all our functions h_j into a function H^{-1} such that

$$H^{-1}(c) = h_j(c) \quad \forall c \in X_j$$

we define an inverse homomorphism (the exponent emphasizes it is an *inverse* homomorphism). On the other hand, by means of the representation in tuples we apply an homomorphism H to a DFA. The composition of the two is, of course, the original DFA.

2.3.3 The look for an effective Homomorphism

In order to find a description for $H^{-1}(c)$, we test the following possible “bit-friendly” definition for $h_j(x)$:

1. $h_j(x) = (p_j \times x + q_j) \bmod m_j$
2. $h_j(x) = (p_j \text{ AND } x) \bmod m_j$
3. $h_j(x) = \text{popcount}(x \bmod m_j)$

These possible definitions are applied (with parameters p_j, q_j, m_j varying from 1 to 256 because x itself is a byte) to DFAs that recognize real data-sets (the ones shown in sec.2.3.6).
In each test, we start by looking for a function that provides a single result in a given subdomain S_j; if none is found, we look for 2 results and so on. Once we find such a function, we follow the “definition” of $h_j(x)$: we check if it outputs the same value inside and outside a subset S_j, that is we check for the following condition:

$$\{x_i = h_j(c_i) : \forall c_i \notin S_j\} \bigcap \{x_j = h_j(c_j) : \forall c_j \in S_j\} \overset{?}{=} \emptyset \tag{2.3}$$

If the condition is not verified (the intersection is not empty), we drop the function and change the parameter again, as described by the pseudocode in algorithm 4. The algorithm can either finish the computation because it finds a good function (i.e.: the return value is FOUND) or fail. The failure happens if no combination of the parameters $\{p_j, q_j, m_j\}$ produces a function h_j whose image set X_j has less than C_{max} elements.

Algorithm 4 Pseudocode for the search of function $h_j(x)$

```
 1: for $\{p_j, q_j, m_j\} \leftarrow \{0,0,0\}, \{255,255,255\}$ do
 2:     for $a \leftarrow 1, C_{max}$ do
 3:         for all $c_j \in S_j$ do
 4:             Compute the set $X_j = \{x_j = h_j(c_j)\}$.
 5:         end for
 6:         if $Card(X_j) > a$ then
 7:             Try with a larger Cardinality $a$, goto 2
 8:         end if
 9:         for all $c_i \notin S_j$ do
10:             if $h_j(c_i) \in X_j$ then
11:                 Try another function, goto 2
12:             end if
13:         end for
14:         return FOUND with parameters $\{p_j, q_j, m_j\}$.
15:     end for
16: end for
17: return FAIL.
```

The results show that, for practical values of the parameter C_{max} (i.e.: $_{max} \leq 64$), only $h_j(x) = (p_j \text{ AND } x) \bmod m_j$ does not cause the algorithm to fail. Moreover, it turns out that $m_j = 255$ in all the tests. Therefore we can define $h_j(x)$ as a simple AND operation with a bitmask:

$$h_j(x) = x \text{ AND } p_j \tag{2.4}$$

Such an outcome has a number of advantages: the number of parameters is limited to 1 (i.e.: small memory footprint), the operation is one of the most basic logic operation (i.e.: it costs a simple logic gate in an hardware implementation and it is very fast and parallelizable if our aim is a software engine) and the definition of $h_j(x)$ is amenable to be described by means of a tree, which means we can redefine each state transition set in a Longest-Prefix-Matching (LPM) description. Finally such a description is always achievable: even in the worst case (all characters produce a different transition and have a different tuple) the correctness of the scheme is not affected. In the following sections , we walk through the properties of such a representation and provide optimizations for our scheme. However, the main advantage is the possibility to concatenate two or more $h_j(x)$, such that we easily obtain a k-step DFA. As an example, if $\{h_0 : X_0 : n_1\}$ describes a transition from state n_0 to n_1 and $\{h_1 : X_1 : n_2\}$ is a transition from n_1 to n_2, then it it straightforward to verify that we the transition from n_0 to n_2 of the corresponding 2-step DFA is as simply as : $\{h_0||h_1 : X_0||X_1 : n_2\}$, where $h_0||h_1$ indicates the concatenation of h_0 and h_1 and $X_1||X_2$ is defined as:

$$X_1||X_2 = \{a_1||a_2 : \forall a_1 \in X_1, \forall a_2 \in X_2\} \tag{2.5}$$

Therefore all we have to take care of is the cardinality of the image sets X_j, that determines the memory requirements for this representation.

2.3.4 Optimizations

In the following we describe the advantages and the properties of the bitmask definition for $h_j(x)$ and elaborate upon the problem of minimizing the cardinalities of the image sets.

2.3.4.1 Permutation for LPM

A first observation on subsets S_j is that, in some cases, they may not be contiguous, i.e.: they may be the union of two or more non-contiguous subsets of symbols. Of course this is detrimental to our mission to minimize the cardinality of X_j. We solve this problem by introducing a *permutation* of symbols: we define a translation table (since we are dealing with bytes, it is a small 256 bytes table that does not increase the cost in terms of external memory accesses) that moves symbols in order to make non-contiguous subsets as contiguous as possible. Finding the optimal translation table is a complex issue since we can define a single translation table for the whole DFA, while subsets may vary from state to state. The good news is that in practical DFAs the number of different subsets is very limited. The bad news is that, as subsets vary from state to state, it may happen that a certain symbol occurs in different subsets. Therefore finding the optimal translation table is an *NP-complete* problem as it is equivalent to the *weighted maximum set packing* problem[39]: we want to find a set-packing (a collection of disjoint subsets) that maximize the total weight of its subsets.
Such weight (w) must take into account the memory impact of the subsets (a large and frequent subset has high utility because, once we put it in the translation table, it is likely to be described by a single bitmask). Therefore w is defined as the product of the cardinalities of subsets and the number of times those subsets appear in the whole DFA.
We attack the problem with the *Co-occurrence Permutation* algorithm, which is based on the co-occurrence of symbols in subsets. First, it computes the character co-occurrence matrix $A^{(0)}$, where an element $a^{(0)}_{i,j}$ represent the number of times characters i and j appears in the same subset multiplied by the cardinalities of the subsets they appear into (such that we replicate our weight metric). Then, the algorithm aggregates all 256 characters in 128 pairs, by grouping characters that present the largest co-occurrence, as depicted in the example of fig.2.12. After that, a new co-occurrence matrix $A^{(1)}$ is computed for all the 128 pairs. Again, pairs are aggregated thus forming 4-characters groups, another $A^{(2)}$ matrix is computed and so on. Therefore the algorithm recursively aggregates characters in a tree and the last matrix $A^{(8)}$ actually collapse into a scalar. Of course we have to define the co-occurrence of groups: for instance, given two symbols pairs i, j and l, m, we can define the pairs co-occurrence as:

- $\frac{1}{4}(a_{i,m} + a_{i,l} + a_{j,l} + a_{j,m})$
- $max(a_{i,m}, a_{i,l}, a_{j,l}, a_{j,m})$

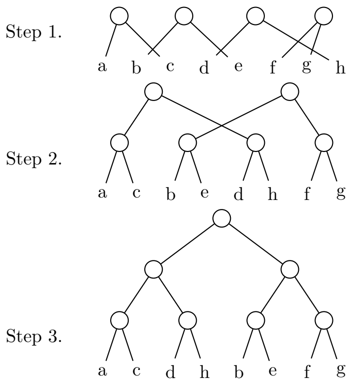


Figure 2.12: An example of *Co-occurrence Permutation* for 3-bit characters

- $min(a_{i,m}, a_{i,l}, a_{j,l}, a_{j,m})$

In our tests, we used the last option ($min(.)$) as it showed best results on all datasets. Finally we put the symbols in the leaves of the tree into a table and the translation table is simply the inverse permutation of such a table. Now, by means of *Co-occurrence Permutation*, subsets S_j can be described with single bitmasks. Then we can use a Longest-Prefix-Matching description of state transition-set and enlarge the number of ideas we can exploit for efficient implementation of DFAs, either taken from the widely studied field of IP lookup or newly proposed.
The effectiveness of the proposed schemes is measured by the total number of $h_j(x)$ we find for all states, once we permute the characters, as such a value represents the "cost" of our h-DFAin terms of transitions. The closer this number gets to the volume of the DFA graph (the cardinality of the edge set), the better the permutation scheme works. The results are shown in fig.2.13: *Co-occurrence Permutation* always gave good results, reaching, for many datasets, the minimum number of transitions or a value very close to it.

2.3.4.2 Bitmap trees

As described above, thanks to the *permutation*, we can define each state by means of LPM structures, such as trees. The adoption of trees is twofold useful: it reduces the memory footprint of the bitmask description and it provides us with another faster way to compute the bitmask parameter in (2.4). As for the latter issue, it is straightforward to see that computing p_j and X_j (the result of h_j on subset S_j) can be now simply demanded to the creation of a tree of all the 256 possible values of the symbol (where last level leaves point to next state) and its subsequent pruning. Therefore, to store our tuples representation (2.2), we can simply use a bitmap tree. However, this does not preclude *permutation*; on the contrary, it takes advantages from the use of a permutation algorithm, because if the characters of same subsets are close to each other, they most likely produce short branches in the tree.
In order to construct a bitmap tree representation of a state, for each character c we get the next state s_{next} and add c in a tree, such that the leaf points to s_{next} as shown in fig.2.14 (where next states are 1, 2 and 3). Once we observed all the symbols, we

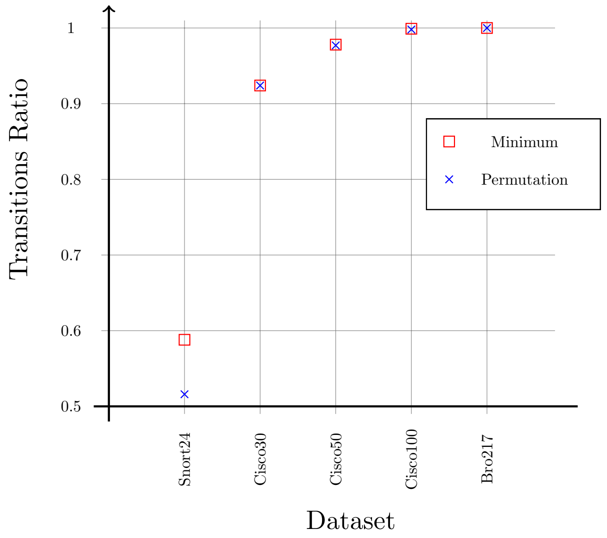


Figure 2.13: Ratio of transitions stored when *Co-occurrence Permutation* is used compared with the minimum number of transitions. The ratio is computed with respect to the case when no permutation is adopted.

prune the tree: if both children of a node x point to the same next state, x inherits children's pointer and children are removed. Finally, we can also remove from the tree the subset described with the largest number of leaves, as it can be stored as a "default transition" to be taken when no match is obtained. In the example of fig.2.14, we remove the leaves pointing to state 1 as they are the most frequent.

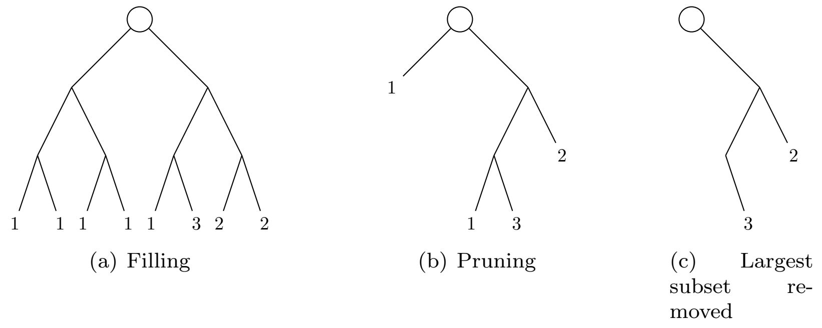


Figure 2.14: An example of state construction in h-DFA for 3-bit characters. The numbers on the leaves are pointers to next states

2.3.4.3 The overall algorithm

Here we retrieve the pieces decribed in the previous paragraphs and finally compose our algorithm for the creation of bitmask-based (or LPM) DFAs. The first step of the algorithm is the computation of subsets and of a series of functions $h_j(x)$ that can define an inverse homomorphism. Then we add a translation table by adopting *Co-occurrence Permutation*, and then we can simply compute an LPM description of each state of our DFA. Notice that an LPM description requires rules be stored in order in (2.2) or in a bitmap tree.

2.3.5 The k-step DFA

As described earlier in sec.2.3.3, the homomorphic (or LPM) description allows for a simple yet memory-efficient computation of k-step DFAs. The algorithm for the creation of a k-step h-DFA is shown in alg.5: it is based on a recursive procedure *compute_1-step* that takes a k-step h-DFA D' and a 1-step h-DFA D and computes the $(k+1)$-step h-DFA D''. As shown in the pseudocode, we add transitions defined by the concatenation of functions $h_1||h_2$ as defined in 2.3.3. Such a concatenation of functions may as well be seen as a concatenation of trees.

Algorithm 5 Pseudocode for the creation of a k-step DFA

```
procedure compute_1-step($D, D'$)
 1: for all state $s \in$ DFA $D$ do
 2:     for all next state $s_1$ of $s$ do
 3:         for all next state $s_2$ of $s_1$ in $D'$ do
 4:             Add_transition($D''$,$s$,$s_2$, $h_1||h_2$);
 5:         end for
 6:     end for
 7: end for
 8: return $D''$
procedure compute_k-step($D$)
 1: for $i \leftarrow 1, k$ do
 2:     $D' \leftarrow$ compute_1-step($D, D'$)
 3: end for
 4: return $D'$
```

2.3.6 Results

The experimental runs have been performed on data sets of the Snort and Bro intrusion detection systems and Cisco security applications [15]. Such data sets, presenting up to hundreds of regular expressions, have been randomly reduced in order to mimic the size (in terms of DFA states and regular expressions) of other sets used in literature [31][17][19][30], as a fair comparison. The characteristics of the data sets are summarized in table 2.2, since we adopt the same datasets of previous sections. As for a construction timing evaluation, our preliminary code always required less than 2 minutes for each DFA to compute the corresponding h-DFA on a Pentium 4 machine and always achieved a successful construction. The regular expressions in the data sets are given as input to the *regex-tool* [36], that produces the corresponding standard DFAs. Such DFAs are, in turn, used as start-point for our algorithms.
Tables 2.7 display the percentage of transitions (and memory) reduction for the different 1-step algorithms with respect to data sets representations through a standard DFA. h-DFA achieves a compression degree which is comparable to the other algorithms while requiring a single memory access per state. Because of its orthogonality

(a) Transitions reduction (%). For δFA also the percentage of duplicate states is reported.

Dataset	D^{2}FA		BEC-CRO	δFA		h-DFA	
	$DB=\infty$	$DB=2$		trans.	dup. states	No Perm.	P_{Co-Occ}
Snort24	98.92	89.59	98.71	96.33	0	94.05	96.52
Cisco30	98.84	79.35	98.79	90.84	7.12	91.28	91.96
Cisco50	98.76	76.26	98.67	84.11	1.1	90.47	90.75
Cisco100	99.11	74.65	98.96	85.66	11.75	87.81	87.93
Bro217	99.41	76.49	99.33	93.82	11.99	78.48	78.48

(b) Memory compression (%).

Dataset	D^{2}FA		BEC-CRO	δFA + C-S	h-DFA + C-S	
	$DB=\infty$	$DB=2$			No Perm.	P_{Co-Occ}
Snort24	95.97	67.17	95.36	95.02	84.16	90.73
Cisco30	97.20	55.50	97.11	91.07	87.91	88.87
Cisco50	97.18	51.06	97.01	87.23	83.82	84.31
Cisco100	97.93	51.38	97.58	89.05	81.63	81.82
Bro217	98.37	53.00	98.23	92.79	69.02	69.02

Table 2.7: Compression of the different 1-step algorithms in terms of transitions and memory.

with other schemes, this is a very appealing result. By using *Co-occurrence Permutation*, we are able to obtain good transitions and memory savings.
Fig.2.15 completes the comparison among 1-step algorithms by showing the mean number of memory accesses per character. D^{2}FA always requires the largest amount of memory accesses, while h-DFA requires always a single access (as D^{2}FA).
However, the main results are shown in tab.2.8, where the memory reduction obtained by computing 2 and 3-step h-DFA according to sec.2.3.5 are reported. In those results, h-DFAis combined with *Char-State compression* 2.1.4 to reduce the number of bits required to store transitions (notice that those techniques 2.1.4 do not add memory accesses). The memory reduction is computed with respect to a standard representation of 2 and 3-step DFA (respectively with 256^2 and 256^3 transitions per state). The compression percentages are very high. Certainly, 3-step h-DFAs still require too large amounts of memory (in the order of tens or even hundreds of megabytes). However, considering the consumptions of standard 2 and 3-step DFAs, which reach even hundreds of gigabytes, our solution is still appealing for DRAM storage. In fact, h-DFArequires at most 10 megabytes to represent 2-step DFA and (in some cases) less than 100MB for a 3-step, thus offering a great speed-up without the unfeasible memory requirements of standard DFAs. Moreover, as our technique is orthogonal to other schemes, we believe that a combination of different compression schemes can reach higher speed-ups requiring less amount of memory.

2.4 Sampling techniques to accelerate regular expression matching

The previous works which propose acceleration techniques 2.3.1 multiply the amount of bytes (strides) processed per cycle. The obvious problems which arise is memory

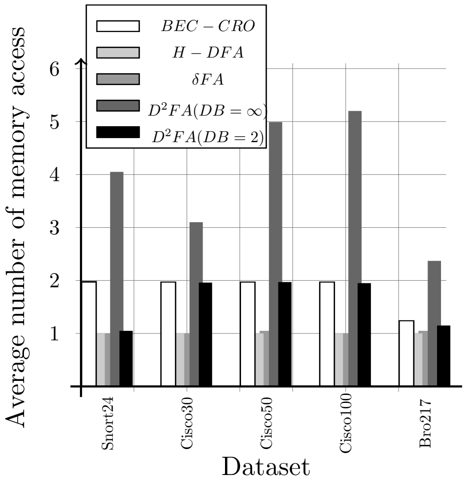


Figure 2.15: Mean number of memory accesses per character.

Dataset	2-step h-DFA + C-S		3-step h-DFA + C-S	
	mem.	trans.	mem.	trans.
Snort24	88.48	98.35	99.34	99.69
Cisco30	98.73	99.50	99.87	99.99
Cisco50	97.84	99.07	99.81	99.92
Cisco100	95.67	98.06	99.42	99.56
Bro217	96.97	95.57	97.98	99.08

Table 2.8: Memory and transition compression (%) for 2 and 3-step h-DFA + *Char-State compression*

blow-up (essentially due to the exponential growth of edge numbers with the stride size) and can be partially mitigated through smart coding for the transition table, alphabet-reduction and default transition compression.
Our approach to the finite automata speed-up is completely innovative: sampling the text, thus having less symbols to be processed. Clearly, sampling introduces some issues; in details, particular automata for the processing are required and a certain probability of false alarms is introduced. We address these issues by the combination of a "sampled" and a "reverse" DFA, two different versions of the original automaton. We perform a first fast search on the traffic through the sampled DFA, which is able to exclude most part of non–malicious traffic, and, if necessary, a more accurate processing through the reverse DFA is triggered, in order to confirm a match. While other works [40] in the area of intrusion detection already show how to sample *messages* to reduce the amount of messages to be processed in a distributed system, the application of sampling to regex-matching is a novelty and one of the main contributions of this work.

2.4.1 Sampling DFAs

In this section we introduce the motivation for DFA sampling and describe the main concepts by means of an example. Moreover, we provide a taxonomy to distinguish the different ways we can sample a regular-expression set or the corresponding DFA.

2.4.1.1 Motivation

The motivation for this work relies on the following assumptions:

- IDS regex data sets are well-written;
- Regular internet traffic does not match properly written IDS regexes.

Indeed, if regex sets are not poorly written (in our tests we use real and effective IDS signatures from Bro and Cisco security applications) a signature match will occur with malicious traffic only. The main assumption is that the majority of traffic is not malicious. Therefore we can take advantage of the fact that a match is a rare event and speed up the average case (regular traffic).
The idea of DFA sampling is to speed up the regex matching by simply "sampling" the traffic stream: we extract a byte every θ bytes from the stream, where θ is the *sampling period*. The sampled bytes are then used as input to a proper sampled DFA. The outcome is that all regular traffic is processed θ times faster. The price to pay is that this process may introduce false alarms: strings that would not match the original non-sampled regex could match the sampled one. Therefore whenever we have a match in the sampled DFA, we have to process the suspect packet through a regular non-sampled DFA.
It is worth noticing that we aim at reducing the number of memory accesses to the main memory that stores the state-machine, while we cannot reduce the number of accesses to the memory where the packet is stored. The reason is that even if we were interested in, say, a byte every two, memories would allow accesses in minimum sizes of k bytes long words. However, in cached-systems, this is an advantage when performing the second stage to check for false alarms: all the memory accesses for this second stage will result in cache-hits, thus reducing the cost of false alarms.

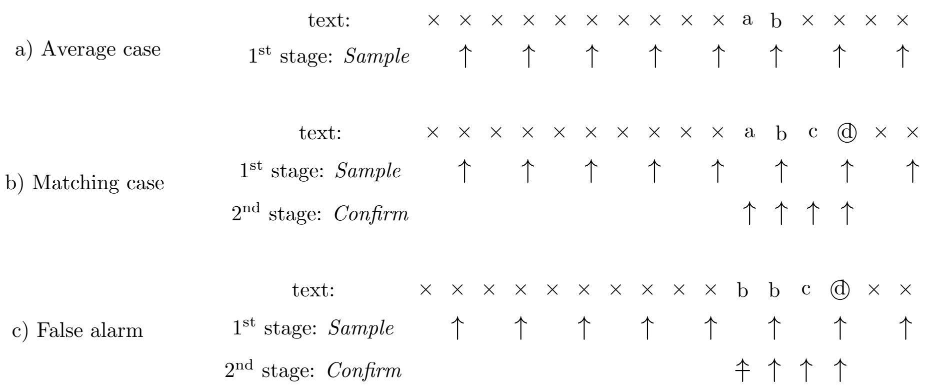


Figure 2.16: Examples of sampling with $\theta = 2$. The regex to match is $ab.*cd$, the sampled one is $[ab].*[cd]$ and the text consists of 16 bytes. Arrows point to observed chars. Sampling performs 12 memory accesses in case of a real match(b) or false alarm(c) or even 8 in the average non-matching case(a). In (c) the striked arrow point to the non-matching char.

2.4.1.2 A Motivating Example

Fig. 2.16 shows the principles of our scheme, with the example regex $ab.*cd$. We use a sampled DFA (that matches $[ab].*[cd]$) and a regular non-sampled one. We perform a first check on the text by using the sampled DFA; if we find a match, we move to the second stage.

Fig. 2.16(a) represents the common scenario with traffic that does not match signatures. It is evident, in this case, that the number of memory accesses and operations to be performed is divided by the sampling period.

Whenever the sampled regex is matched (lower two cases in figure 2.16, where the circled letter indicates the sample where we find the match), the non-sampled text has to be checked to confirm the match. To address this issue, the simplest and fastest way (see section 2.4.3.2) is to adopt DFAs that match reversed signatures (in our example, $dc.*ba$). Any regular language is closed with respect to reversing operations [37], therefore we can always reverse a regular expression and match it inside a text by observing the text backwards from the end to the beginning. Then, if a match occurs in the second stage, because of the equivalence of reversed and forward DFAs, we have a confirmed match (fig. 2.16(b)). Otherwise, we can claim that a false alarm occurred (fig. 2.16(c)).

2.4.1.3 Taxonomy of DFA Sampling

Sampling can be performed in a number of different ways. In the following we give a brief description of these techniques and of the models that we use. From the point of view of the sampling period θ, we can have:

1. Fixed Period Sampling (CPS) : θ is constant;

2. Variable Period Sampling (VPS) : $\theta = \theta(s, c)$ is a function of the DFA state s and/or the input character c.

The construction of a sampled DFA can be classified as:

1. Static : $\theta(s, c)$ is decided during construction;
2. Adapting/Evolving : $\theta(s, c)$ evolves adapting to traffic features, reducing false alarms and maximizing speed-up.

However, in this first work on sampling we focus on static constant period sampling.

2.4.2 Regex sampling rules

Here we introduce basic theoretical results on DFA sampling. As in signal sampling theory Nyquist condition is the only one rule to satisfy, also when dealing with regular expressions matching a simple unique condition has to be satisfied to perform a correct sampling:

Lemma 1. *Let DFA A describe a single regular expression $\boldsymbol{R}$[1] and let a text T match $\boldsymbol{R}$. The corresponding sampled DFA $A^{\mathbb{S}}$ will match the sampled text $\mathbb{S}T$ if the sampling period θ satisfies the following:*

$$\theta \leq \min |r| \qquad \forall r \in \boldsymbol{R}$$

Proof. The proof is straightforward. In order to match the sampled text, we have to extract (by sampling) at least a character from the substring of T that matched A. Thus the condition follows. $\square$

Lemma 1 limits the sampling period that can be used when matching a single regular expression. However when working with DFAs that match a set of regular expressions, it still applies, as long as the limit is moved to the minimum length of any string that match any regular expression in the set. Moreover the lemma states that, if the condition is satisfied, we may have false positives but we cannot have false negatives. This important result is the basis of the research presented in the rest of this research.

2.4.2.1 Regex rewriting

The application of sampling can be performed by rewriting regular expressions according to few simple rules. In the following we use the notation:

$$\mathbb{S}^{X_0}_{\{\theta\}} \boldsymbol{a}$$

to refer to the application of sampling to the regular language $\boldsymbol{a}$. In particular, the symbol $\mathbb{S}$ represents the *sampling* operator, $\{\theta\}$ is the series of sampling periods $\theta_0, \theta_1, \ldots, \theta_N$, and X_0 is the position of the first sampled character. In the rest of this section, the sampling operator $\mathbb{S}$ will be adopted also as an exponent (i.e.: $A^{\mathbb{S}}$) to denote the sampled version of a DFA.
Basically, we show the application of the $\mathbb{S}$ operator to four main cases:

[1] Throughout the whole section, bold letters represent regular expression, while non-bold stand for single letters.

1. simple string *str*
2. concatenation of regular expressions $\boldsymbol{a}$ and $\boldsymbol{b}$: $\boldsymbol{ab}$
3. union of regular expressions $\boldsymbol{a}$ and $\boldsymbol{b}$: $\boldsymbol{a}|\boldsymbol{b}$
4. star closure of a character a followed by a regular expression $\boldsymbol{b}$: $a{*}\boldsymbol{b}$

The sampling of a string is straightforward, and it simply consists of extracting characters at the positions defined by $\{\theta\}$ with offset X_0:

$$\mathbb{S}_{\{\theta\}}^{X_0} str = \{str(X_0 + \{\theta\})\}$$

The offset X_0 is critical also when sampling the concatenation and union of regular expressions, it is immediate to show that:

$$\mathbb{S}_{\{\theta\}}^{X_0} \boldsymbol{ab} = (\mathbb{S}_{\{\theta\}}^{X_0} \boldsymbol{a})(\mathbb{S}_{\{\theta\}}^{X_0} \boldsymbol{b})$$

and

$$\mathbb{S}_{\{\theta\}}^{X_0} \boldsymbol{a}|\boldsymbol{b} = (\mathbb{S}_{\{\theta\}}^{X_0} \boldsymbol{a})|(\mathbb{S}_{\{\theta\}}^{X_0} \boldsymbol{b})$$

Finally, a star closure of a character a followed by a regular expression is simply :

$$\mathbb{S}_{\{\theta\}}^{X_0} a * \boldsymbol{b} = a * \bigg|_{i=0}^{\theta-1} \mathbb{S}_{\{\theta\}}^{i} \boldsymbol{b} = a * \mathbb{S}_{\{\theta\}} \boldsymbol{b}$$

We can easily verify the sampling of $a*$ is again $a*$. Then, since $a*$ consists of all the possible n-repetitions of a (where $n \in \mathbb{N}$), the sampling offset we apply to $\boldsymbol{b}$ can be any, hence the big **OR** operator $|$.
Now, although the last case follows from the first three (concatenation and union), it is worth describing it because of its frequent occurrence in real regular expression sets. Indeed, many regular expressions in real IDS/IPS data-sets adopt star closures and most of the times they are unanchored rules (because security signatures may occur everywhere in the text) of the form: $\boldsymbol{.{*}a}$. Therefore sampling produces a union of the regex $\boldsymbol{a}$ sampled with all different possible offsets. As an example, let us suppose we have $. * abcde * fgh$ and we are sampling with fixed period $\theta = 2$. By applying the previous rules, it follows that:

$$\mathbb{S}_2[. * abcde * fgh] = . * (ac|bd)e * (fh|g)$$

2.4.3 Constant Period Sampling

2.4.3.1 First stage: Sampled DFA

As above mentioned, the idea of DFA sampling is to speed up the string matching by extracting a byte every θ bytes from the stream and giving such characters as input to a “sampled DFA” for a first approximate search (to be subsequently confirmed). Regarding the Constant Period Sampling (CPS) case, such sampled DFA can be simply obtained by properly rewriting the regexes and building the DFA according to the new rule-set.

In details, for the process of regex rewriting, we can apply the results of section 2.4.2.1 by selecting a single value θ for all the sampling periods θ_i. Instead, concerning the offset X_0, which is the position of the first character to be sampled in the regex, we have to take into account all the possible starting values. This way, the resulting complete automaton can be used for string searching regardless of the point in which we start to sample the traffic.
The pseudocode 6 just shows the overall procedure for rewriting a regex by using a constant period θ and by adopting all the possible values for the starting offset: we split the regular expressions into sub-elements that can be processed directly by adopting the rules in sec. 2.4.2.1. In order to simplify the code, a pre-processing (not shown here) is adopted to convert "+" closures in "*" (i.e., $a+$ becomes $aa*$) and to take care of the cases when the sampling period is higher than the length of the minimum string between two closures or the presence of unions ("|"). By repeating such a process for all the regexes belonging to the set, we obtain the "sampled" rules on which the "sampled DFA" has to be built. Such a resulting automaton is a simple DFA and does not require additional information on the states or on the transitions. From this observation and as suggested by the results of section 2.4.2, we can claim that a regular language is closed with respect to the fixed sampling operator.
However, even after pre-processing, some regular expressions may still be so short to make sampling unconvenient. For instance $\mathbb{S}_3 abc = [abc]$: although the sampled regular expression is valid, it is only 1-character long, thus potentially yielding a large number of false alarms. The good news is that these extremely short regular expressions are not very frequent. Therefore a good and effective solution is to hard-code them, moving the matching problem from data to code and adopting a function *regex_match(c)* which is basically composed of *switch()* – *case* and *if* – *then* statements. This is a well-investigated idea [41][42][43] that is shown to be very useful with a small number of regular expressions. It is also compatible to our sampling approach: the *regex_match(c)* function can still access all the bytes of the un-sampled text (which are available to the code, as discussed in section 2.4.3.3) thus keeping the processing engine busy between two successive memory accesses to the sampled DFA. Since the number of regular expressions to be matched by such a code is small (as already pointed out, short regular expressions are fairly rare), the whole data set which is necessary for such a code can be kept in the local cache, thus requiring no further accesses to the external memory blocks.

Algorithm 6 FPS of a regex **a** with period θ.

```
procedure sample_regex($\mathbf{a}, \theta$)
 1: $p_{next} \leftarrow$ first_pos($\mathbf{a}$,"*") $-1$
 2: $l \leftarrow \mathbf{a}[p_{next}]$
 3: while $p_{next} \neq$ NULL do
 4:     $\mathbf{b} \leftarrow \varepsilon$
 5:     for all offset $x$ do
 6:         $\mathbf{b} \leftarrow \mathbf{b}|$ sample_str($\mathbf{a}(p_{prev} + x \ldots p_{next})$)
 7:     end for
 8:     $\mathbf{a}' \leftarrow \mathbf{b}l * \mathbf{a}'$
 9:     $\mathbf{a} \leftarrow \mathbf{a} << p_{next}$
10:     $p_{next} \leftarrow$ first_pos($\mathbf{a}$,"*") $-1$
11: end while
```

2.4.3.2 Second stage: Reverse DFA

Whenever in the "sampled DFA" a matching happens (i.e., an accepting state is reached), we have to process the text again, by means of the original non-sampled DFA, in order to obtain a confirmation of the match. As already mentioned, the reason is that sampling a DFA introduces a false alarm probability, since we check only a subset of the characters of the string.
Let us suppose the matching has been detected at the $k-th$ sample in the text. By using the original DFA for such a further search, a problem arises: which byte of the packet has to be the starting point for the matching confirmation processing?
The simplest solution could be processing again the overall packet (i.e., starting from the first byte of packet), but this heightens the processing and yields an excessive delay. Therefore, a more efficient technique could be to "remember" the last time the process has been in the root state (hereafter simply called state 0) and start from the corresponding character. However, even this solution could be too expensive, requiring the processing of a big number of characters, as shown in the following example.
Let us assume our DFA (shown in figure 2.17(a)) matches the two regular expressions:

$$.*ab.*fgh$$

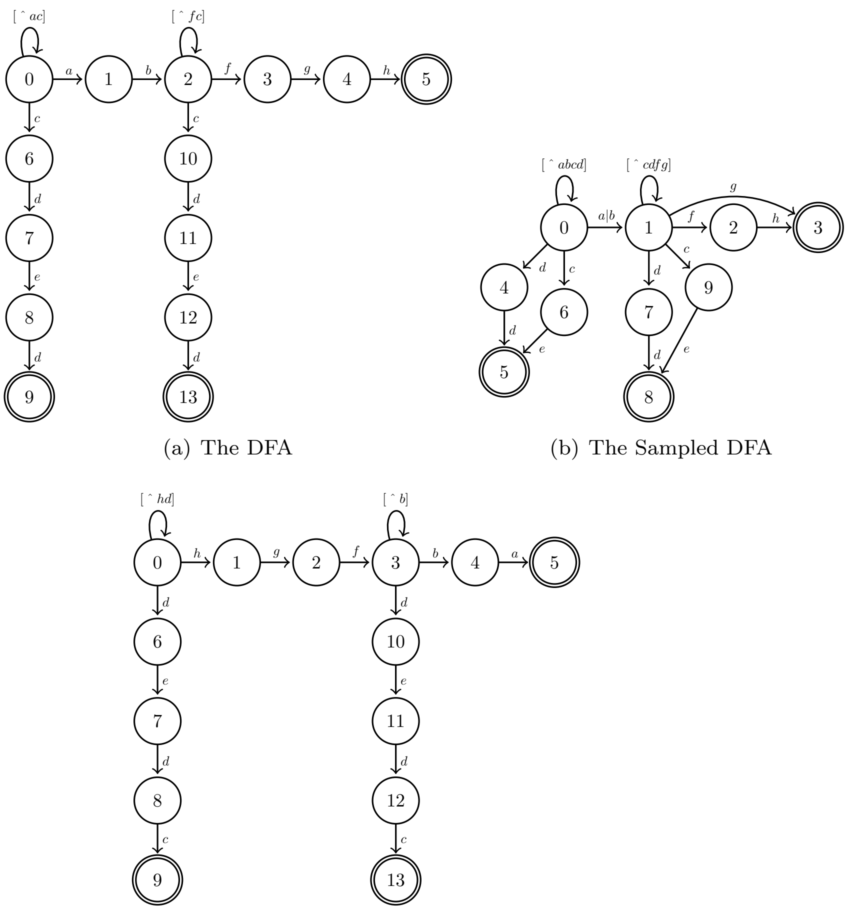
$$.*cded$$

The sampled DFA (fig. 2.17(b)) is built on the sample regexes:

$$.*(a|b).*(fh|g)$$

$$.*(ce|dd)$$

Suppose we read the text: $T = xxabxxxxxxxcdedxxx$ and we sample the text with $\theta = 2$, obtaining the sequence: $T' = xbxxxcexx$. By triggering the sampled DFA with such a sequence, the processing of the character e reports a match in state 8, which has to be confirmed with the standard 1-step DFA (fig. 2.17(a)). The last sampled character for which the sampled DFA was in state 0 is the second x of T. Therefore, if we use a forward DFA to confirm the match, we need to process all the characters from a (position 3) to the last d (position 16): we read 13 characters to confirm a match of a 4-bytes string ($cded$). This is due to the presence of a closure ($.*$) within one of the regex matched by the DFA. Indeed, for each closure, the DFA replicates the states corresponding to some regular expressions (as happens in state 2 in fig. 2.17(a)). This means that we can start matching a whole signature starting from a state which is not the root state 0. This requires the processing of a much larger number of characters than strictly needed.
For these reasons, in order to improve the performance of the second stage, we propose a novel scheme where a *reverse DFA* has to be built. Such a technique requires a slightly larger amount of off-line processing: all the regexes have to be independently reversed and a new DFA has to be built according to such new rules. However, this approach has the advantages that we can start the second stage reverse-matching from the last sample. More precisely, in order to take into account all the characters belonging to the string, the correct starting point for the reverse DFA is the $(k+1)$-th sampled char in the text. The reason is that the sampled DFA may report a match for



(a) The DFA

(b) The Sampled DFA

(c) The Reverse DFA

Figure 2.17: Example of the finite automata needed for sampling (only the forward transitions are shown for readability): (a) is the standard DFA, (b) is the sampled one (with $\theta = 2$) and (c) is the reverse DFA.

a signature that ends between the actual and the next sample. Therefore, we process some useless characters too (the first ones in the text after the matched string), but this does not affect the detection of the string by the reverse DFA. On the contrary, since the match in the sampled DFA may occur some character before the real match of the non-sampled string, by moving one sample further we ensure the correctness of the scheme at the cost of processing a few more (less than θ) characters than the strictly needed.
Thus, if we adopt a reverse DFA (fig. 2.17(c)) in the previous example, and we have a match in the k-th sample, we start the reverse DFA from the next sample (the $(k+1)$-th) and go backwards, processing the substring $T'' = xdedc$ (i.e., the reverse of the substring $cdedx$ from T). As easily verifiable, the reverse DFA correctly confirms the match by processing 5 characters only, while the forward DFA needed 13 bytes. Notice that, in this example, if we started the reverse DFA from the matching sample (character e) we would wrongly miss the match.
In the confirmation stage performed through a reverse DFA, we confirm a match whenever an accepting state is reached (notice that an accepting state in such a DFA represents the beginning of an original non-reversed regex). Instead, we can immediately detect a false alarm and stop our search whenever we return to the state 0 with any character belonging to the subset of positions $0 \dots k \times \theta$. Indeed, the characters not belonging to the string, which are between the $(k \times \theta + 1) - th$ and the $((k+1) \times \theta) - th$ char, could trigger a return to the state 0, but this does not imply a false alarm. Instead, if the return to the state 0 is forced by any character from the $(k \times \theta) - th$ to the first one (which certainly belong to the probable matching string), then a false alarm is detected.

Further performance refinement However, in some cases, locating the $(k+1)$-th sampled char in the text is not a good choice for the performance of the correct matching search. An example is shown in fig. 3 where the sampled DFA (with $\theta = 2$) for the regular expression $abc * d$ is depicted. With such a DFA, when we process the text $T = xxxabcccccccccdx$ we trigger a reverse DFA confirmation for each of the sampled character c inside the text, always reporting a false alarm. When, at last, we read the character d, we restart the final and conclusive confirmation stage with the reverse DFA (with a right positive result). Notice that this problem occurs because of the closure on the accepting state of the DFA and affects the performance of the technique solely, while the correctness is not invalidated. A simple solution to this problem is to start the confirmation process when leaving a matching state only (i.e., state 2 in the example). In this way, we make sure that the reverse DFA has really started matching a regular expression.
The pseudocode for the lookup is shown in alg.7. In the listing, s represents the actual state, s_{next} is the next state and i and j are the text position we currently read respectively for the sampled DFA $A^{\mathbb{S}}$ and the reverse DFA A^R (this convenient exponent–notation will be adopted hereafter). In the pseudocode, lines 1-3 and 22-24 are part of the regular sampled DFA walk. Line 4 represents the condition we discussed above: we start a confirmation match only when we leave an accepting state (s.acc is the accepted rule) and move to a state that does not match the previous rule. This takes care of the cases where the accepting state has a loop. In lines 5-8 we initialize and start the first part of the reverse DFA walk and do not care about the occurrence

Algorithm 7 Pseudo-code for the lookup procedure.

procedure lookup (T,$A^{\mathbb{S}}$,A^R,θ)

```
 1: $s \leftarrow 0$
 2: while $i < length(T)$ do
 3:     $s_{next} \leftarrow A^{\mathbb{S}}[s, T[i]]$
 4:     if $s$.acc $> 0$ AND $s_{next}$.acc $\neq s$.acc then
 5:         $s' \leftarrow 0$
 6:         for $j \leftarrow i, i - \theta(s)$ do
 7:             $s' \leftarrow A^R[s', T[j]]$
 8:         end for
 9:         while $s' == 0$ do
10:             $s' \leftarrow A^R[s', T[j]]$
11:             $j \leftarrow j - 1$
12:         end while
13:         while $s' \neq 0$ do
14:             $s' \leftarrow A^R[s', T[j]]$
15:             if $s'.acc > 0$ then
16:                 Confirm Match of $s'.acc$
17:                 return to outer loop
18:             end if
19:             $j \leftarrow j - 1$
20:         end while
21:         claim False Alarm
22:     end if
23:     $i \leftarrow i + \theta(s_{next})$
24:     $s_{next} \leftarrow s$
25: end while
```

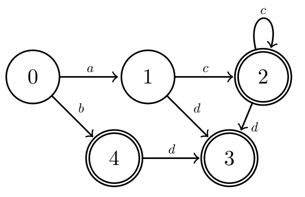


Figure 2.18: Example of a sampled DFA for regular expression: $abc * d$. Only some edges are shown.

of state 0. Then the first while loop (lines 9-12) is in charge of cases where the first state of the reverse DFA has a closure (for instance: $(abcde*)^R = e * dcb$). Finally the next *while* loop performs the reverse DFA walk.

Splitting the reverse DFA As a final comment, we consider splitting the reverse DFA into several smaller DFAs, one for each subset of regular expressions corresponding to a matching state in the sampled DFA. In details, for each accepting state $\underline{s_j}$ in the sampled DFA $A^{\mathbb{S}}$, we observe the subset of regular expressions $X_j = (\mathbf{r_1}, \mathbf{r_2}, \ldots \mathbf{r_k})$ that $\underline{s_j}$ accepts. For each different subset X_j we create a reverse DFA A_j^R. This way, whenever the sampled DFA reaches a matching state $\underline{s_j}$, we start the reverse match with the corresponding A_j^R. This approach reduces the number of steps to perform in the reverse match when a false alarm occurs. Indeed, on a large DFA A^R a piece of text that does not match a given regular expression may match a part of any other regular expression in the set hence keeping the walk away from state 0 and preventing us to claim the false alarm. This does not happen when adopting several small DFAs A_j^R. Moreover this scheme remarkably reduces memory wastage, since, as shown in [28], n DFAs are less expensive than a single DFA for n signatures in terms of number of states and size. However, using n reverse DFAs requires in the sampled DFA additional information which link each accepting state to its own reverse DFA. In the final experiments we will discuss about the performance of both schemes.

2.4.3.3 Possible implementations

The discussion above shows that the basic idea of our approach is to divide the problem into common cases (no matches) and "exceptional" events (a match). To deal with these two cases, we perform two different processing stages. It is worth mentioning that the second stage does not have to be performed necessarily by the same processing engine that executes the first stage. For instance, k_2 processing engines can be allocated for this job, dealing with all the alarms produced by $k_1 > k_2$ first-stage engines. However, while this possible implementation can increase the speed of our solution, in this work we describe our approach as performed by single entities (first and second stages in the same engine), as we are interested in showing a proof of concept rather than the best possible implementation.

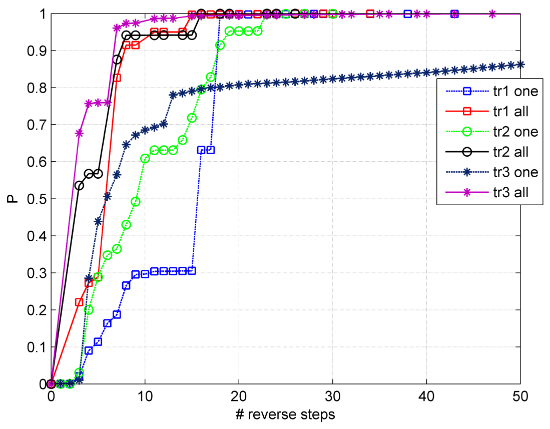
2.4.3.4 Dealing with DoS attacks

Generally, when dealing with security applications, every approach that tries to optimize a frequent case by relying on the assumptions that certain events (for us, a signature match) are relatively uncommon, is subject to be affected by aimed attacks that try to increase the probability of the rare events, thus invalidating the purpose of the method. However, as proposed in [18], such Denial Of Service (DoS) attacks can be taken care of by observing the "behavior" of the incoming flows and distributing them into different queues (with different service rates) accordingly. In our scheme, the "good" or "bad" behavior of a flow is measured by the number of false alarms it generates within a time frame, since false alarms represent the largest portion of the processing cost. Therefore, according to this mechanism, flows that generate a large number of false alarms are sent to the queue with lowest service rate, while "good" flows (i.e.: with few false alarms) are queued and serviced with high rates. However, in this work we do not deal with the details of such an approach, inviting the interested readers to find more details in [18].

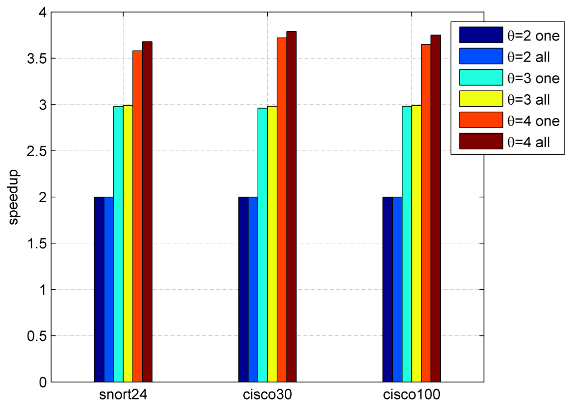
2.4.4 Experimental Results

In order to understand the advantages and costs of our approach, in the following we present the results of a number of experimental tests on real traffic. In details, the purpose of these tests is to show the behavior of the sampling approach in real cases and as the parameters of the problem vary.
To propose verifiable and valid tests, we use:

- the datasets of regexes of real Bro and Snort intrusion detection systems and Cisco security appliances [15];



(a) Required steps in the reverse walk.



(b) Overall speedup.

Figure 2.19: Using an overall reverse DFA (*one*) or one DFA per regex subset (*all*).

- the Michela Becchi's regex tool (which is freely available [36] and proven very stable and powerful) to create the DFAs from the "sampled" regexes.

More precisely, we processed the real regex datasets (where the number in the name indicates the number of regexes) with our tools for creating the new regexes (i.e. sampled and reverse), which are then parsed by the Michela Becchi's regex tool to create the DFAs (which therefore result to be the sampled and reverse DFAs).
We dumped several traffic traces from our department network. Such traces were composed by several flows, associated with different kinds of applications (peer-to-peer, web browsing, multimedia, ftp), therefore they encompass a realistic mix of both mainly textual streams and binary streams. The different TCP connections have been reassembled by using TCPflows [44] and the resulting streams have been concatenated in order to obtain the overall traces. The first runs aim at comparing the efficiency of using an overall reverse DFA for all the regexes (*one* in figure 2.19) or a single reverse DFA for each subset of regular expressions corresponding to a matching state in the sampled DFA (*all* in figure 2.19). The graph in figure 2.19(a) shows the number of steps required in the reverse walk by using the two different techniques when processing three real traces (of length 52MB, 48MB and 66MB respectively) with Cisco100 as regex databases. Instead, figure 2.19(b) illustrates the speedup (computed as ratio between the trace length and number of accesses required) when processing the firs trace (52MB) for different regexes datasets. As foreseen in section 2.4.3.2, using one reverse DFA for each subset of regex reduces the number of steps to perform in the reverse walk when a false alarm occurs and allows higher speedups, along with a memory saving.
To compare our sampling scheme with a classical DFA which processes all bytes, we took as reference hardware platform the Network Processor Intel IXP2800 [45]. Network Processors offer very high packet processing capabilities (e.g. for gigabit networks) and combine the programmability of general-purpose processors with the high performance typical of hardware-based solutions. In particular the IXP2800 is designed to perform a wide range of functionalities, including multi-service switches, routers, and broadband access devices. It is a fully programmable network processor, characterized by a hierarchy of processing units (a XScale core and 16 32-bit micro-engines MEv2, running at 1.4GHz) and memory devices (4KB of local memory, 16KB of scratchpad memory, 256MB of external SRAM and 2GB of external DRAM). The bigger the memory, the slower the access to it.
We simulated the functioning of our algorithm by putting the automata in DRAM, given the large memory required by standard DFAs, and reserving for pattern matching 4 microengines with zero-overhead full threading support (i.e., 8 threads per microengine with no penalty for context switch). Consider that each state traversal requires a DRAM access, as well as the readings of packets, and that, in turn, each DRAM access costs on the average 1270 clock cycles. Fig. 2.20 reports the results in terms of bit rate when processing a real trace of 20MB with a common DFA (i.e. $\theta = 1$) or with our sampling scheme (with $\theta = 2, 3, 4$). It is evident the speedup of sampling DFA, which allows to multiply the bit rate. The payment due to the check in the reverse DFA is very slight because of the low occurrencies of false positives in real traffic. Therefore, the sampling DFA enables a big saving of processing, according to the sampling period, which results in a higher sustainable bit rate.
In order to perform comparisons between our solution and the more efficient schemes

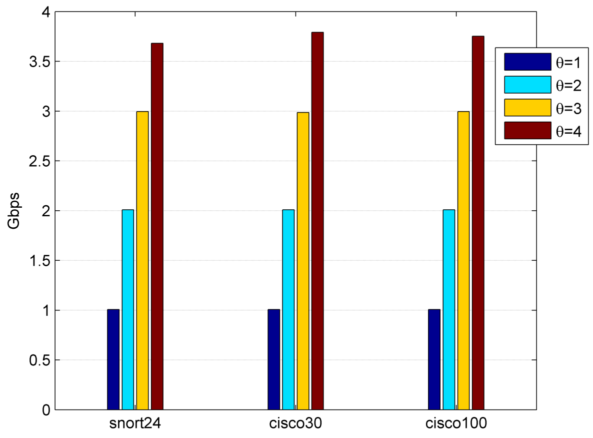


Figure 2.20: Bit rate with a standard DFA ($\theta = 1$) and sampled DFAs ($\theta = 2, 3, 4$).

for speeding up the matching search in DFAs, we implemented the techniques proposed in [38]. In details, we apply alphabet-reduction and default transition compression to k-DFAs introduced in [38], which in turns are based on D^2FAs.

The graph in figure 2.21 shows the data rates achieved when processing real traces of 52MB ($trace1$) and 48MB ($trace2$) and by adopting Cisco100 as dataset. In particular, we compared our scheme and the one implemented according to the directions in [38] by setting $\theta = k$, where the former represents the sampling period and the latter the stride length (i.e., the amount of bytes which are processed at each step in [38]). Notice that the runs pointed out that both the schemes detect the overall number of attacks in each case (i.e. for each mix of traces and databases). The 1st and 3rd bars of the histogram represent the bit rates for our sampling scheme, while the other two bars report the values for the multi-stride scheme. The advantages of our schemes are clear.

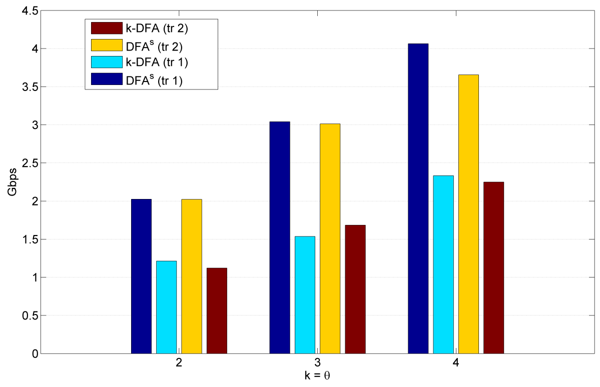


Figure 2.21: Bit rate with k-DFA and our DFA^s.

As for the memory size requirements, we do not report the results because, since our technique produces regular DFAs, it can actually be coupled with many of the compression schemes proposed in literature (and cited in the previous introductory sections) thus avoiding memory blowup. However, it is important to point out that the sampling operation does not overly increase the size of the DFAs. Another experiment aims at describing the effect of the features of regular expressions on the number of false alarms (and hence speed) in the sampling approach.

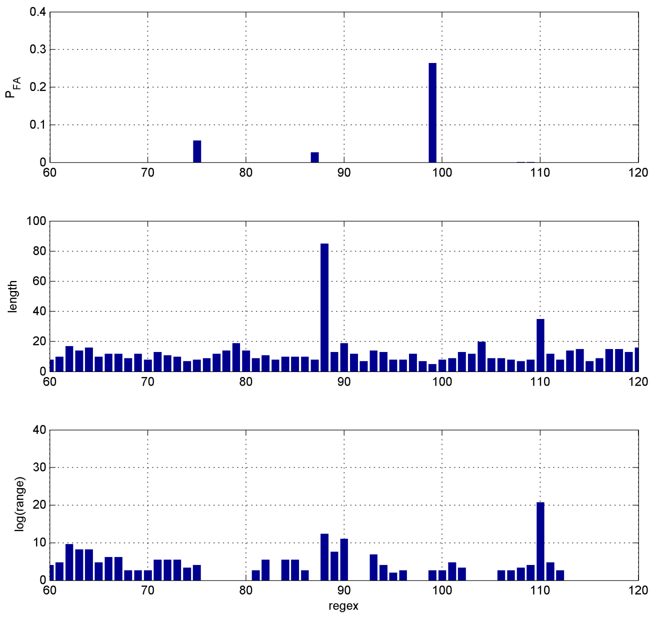


Figure 2.22: False alarms, length and range for each signature.

In fig. 2.22 we report three aligned graphs that correlate the measured probability of false alarms (P_{FA}, top graph) generated by each signature to the signature length (graph in the center) and the signature range (bottom graph). The signatures are labeled by the numbers on the x-axis and the graphs represents the results of a sampled match of the Cisco200 regular expressions with $\theta = 4$. The length of a signature is defined as the length of the shortest string that matches the signature, while we define the range as the cardinality of the set of all strings matching that signature. Of course, closures ($*$ or $+$) would cause the range of a signature to be infinite (remember closures represent the unlimited repetition of a character). For this reason, in order to properly represent ranges in the graph, we set the range of a closure to a large number (10000). The figure shows that a few short signatures are responsible for the majority of the false alarms. In the following, from the inspection of a few distinct cases, we show how to extrapolate the general behaviour. Signature 99 contributes to more than 25% of all false alarms, this is mainly due to its short length and to its fairly regular range. On the other hand, signature 110 does not produce any false alarm because, even if it has a large range, it has a remarkable length. Signature 115, instead, is quite short (the shortest length bar in the middle graph), but it does not contribute to false alarms because of its very limited range. These examples justify the intuitive idea that short signatures with large range are the most likely to provoke false alarms. Another comment is that the length of a regular expression has a larger effect on false alarms than its range.

2.5 Counting Bloom Filters for pattern matching and anti-evasion at the wire speed

Standard pattern matching methods used for deep packet inspection and network security can be evaded by means of TCP and IP fragmentation. In order to detect such attacks, Intrusion Detection Systems must reassemble packets before applying matching algorithms, thus requiring huge memory and a large amount of time to respond to the threat. To reach very high speeds, finite automata in FPGAs (Field Programmable Gate Arrays) are used; however, for adding and deleting strings, a hardware reconfiguration is required, which is too expensive. The simplest approach is a TCAM (Ternary-Content Addressable Memory), which stores all the strings: searches can be very fast (a single clock cycle) but the high cost makes TCAM infeasible for large signature sets.

2.5.1 Related works

Recently, Bloom Filters (BFs) have also been used for pattern matching [46] [47]. They are hash-based structures which trade a certain degree of accuracy for considerable memory savings. BFs were born to represent a set of elements and to perform membership queries, so they can be adopted for pattern matching by simply constructing filters according to a set of signatures. The advantages are the compact representation typical of BFs and a remarkable reduction of the amount of traffic handled by the slow path, which result in a general performance improvement and scalability of IDSs. The work in [46] adopts parallel BFs: each of them represents the strings of a specific length, in this way allowing for a fast search. The work in [47] instead combines BFs and parallel hashing: packets are first passed through a BF which detects some strings by acting as accelerator. Such strings are then dispatched to the parallel hashing engine, which performs a hash comparison, and, in case of a hash hit, compares the input string to the actual string to eliminate any false positives. A brief introduction to Hash and Bloom Filters could be found in appendix.
Moreover, while the set of signatures to be detected changes very frequently (due to the continuous creation of new viruses and attacks), Bloom Filters do not address the issues of changing items in a set, hence Counting Bloom Filters (CBFs) have been designed. They are based on the same principles of BFs but use counters to take into account the occurrences of items, in this way allowing for quickly updating the string set and candidating to be used in pattern matching. The work in [48] proposes the use of a bit vector in which each bit corresponds to a counter of the CBF representing the string set. Whenever a member is added to or deleted from the CBF, the corresponding counters are incremented or decremented, respectively. If a counter changes from zero to one, the corresponding bit in the bit-vector is set,

while it is cleared if the counter changes from one to zero. Since the counters are changed only during addition and deletion of strings in the set, and these updates are relatively less frequent than the actual query process itself, the authors suggest the CBF should be maintained in software and the corresponding bit vector in hardware, thus saving memory resources. However, several research efforts [49] [50] show how to evade standard pattern matching techniques by splitting into several packets or by slightly changing (e.g., by UTF-8 synonyms) the malicious strings, thus making useless the pattern matching on single packets. Many software tools (e.g., FTester, FragRoute or Nikto) even implement such evasion attacks. This work focuses on the "fragmentation" evasion (and through all the section we simply use the term *evasion* to refer to it). Currently, the only way to deal with this problem is to reassemble the overall packet flows and afterwards apply standard pattern matching algorithms. This dramatically increases requirements for security systems in terms of both memory and processing power, especially for securing traffic at wire speed. Moreover, to face few malicious flows, an IDS must reassemble all passing flows. In [49] is shown that the processing for TCP reassembly can be remarkably reduced by optimizing for the expected case when most TCP segments are in order. However, the costs for both memory and processing remain too high.

Some work tries to avoid the need for flow reassembly to detect evasion attacks. The authors of [51] propose an architecture composed of a Flow Processor and a Payload Processor. The former maintains per flow state information for multi-packet signature detection, while the latter uses a combination of parallel BFs. More precisely, the Payload Processor adopts, for each length, a BF which represents all the strings of that length, as well as a BF which represents all the string pieces of that length. When a packet arrives, a complete check is performed on all the filters (which is an expensive process). If a match is detected, the flow database is updated and the state becomes malicious (if a whole signature is found) or suspicious (if a simple piece is found). Whenever the flow state is malicious, the flow is passed to an analyzer for a further deterministic check. This scheme assumes that packets are not ambiguous, in order and not overlapped, thus neglecting many real issues. Moreover, the use of filters for prefixes of 1 or 2 bytes appears too expensive for memory requirements, processing power and alert rate, thus making such a system not efficient.

The basic idea of [52] is to split the signatures to be searched by pattern matching algorithms into small substrings. In this way, if a sufficiently large piece is completely inserted in a packet, it is easily detected. Otherwise, the attacker is forced to use several very small or out-of-order packets, and such abnormal behaviors are revealed by adopting proper heuristics. Both techniques are performed in the fast data path, thus guaranteeing a big saving in terms of both time and memory with respect to the overall flow reassembly. This solution, though, presents some weaknesses: when the counter of small or out-of-order packets of a specific flow exceeds a threshold, such a flow is diverted to the slow data path. The paper claims that this threshold is set according to the signature length which the small packets belong to. Unfortunately, it is not possible to know such a parameter before the flow has been reassembled and the entire signature has been detected, hence this heuristic appears difficult to be used.

Anti-evasion is a hard problem and to find a conclusive solution is very difficult. Hence, our work is an attempt to address the problem in an alternative and effective

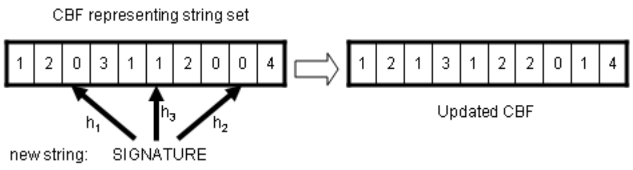


Figure 2.23: Addition of a new string in a CBF.

way, thus creating new opportunities for future research on this topic. The main target is the same of [51] and [52]: avoid flow reassembly for detecting an evasion attack. We will first show how CBFs can be used for anti-evasion techniques, thanks to their appealing features in terms of compactness and speed, update capability and emptying feasibility. Then, a comprehensive CBF-based solution for anti-evasion will be illustrated.

2.5.2 CBFs for pattern matching and anti-evasion

BFs do not allow changes in the item set. In fact, deletion cannot be done by simply changing ones into zeros, as a single bit may correspond to multiple elements. Therefore, CBFs have been introduced, which are based on the same idea of BFs, but use fixed size counters (also called bins) instead of single bits of presence. When an item is inserted, the corresponding counters are incremented; deletions can now be safely done by decrementing the counters.
CBFs are used in representing elements for their well-known compactness and speed. Our idea is to use them also for pattern matching and anti-evasion, due to their innovative capabilities of quickly updating the set they represent and counting the occurrences of elements. The first property can be used to rapidly take into account each new virus definition, with no need to rebuild the overall structure: in order to add a new string to be matched, it is sufficient to apply the hash functions to such a string and increment the proper CBF bins (as shown in figure 2.23, where the new string SIGNATURE is added and h_i are the hash functions). The same principles can be used for removing an obsolete string.
Counting the occurrences of elements makes CBFs appealing just for anti-evasion. Indeed, a CBF can be set to represent the different substrings composing a string: the arrival of any pieces belonging to the string triggers a decrease of the proper bins and when the filter is completely reset to zero, the overall match is detected. In this way, in order to reveal an evasion attack, it is no longer necessary to divert a flow to a slow data path engine, which must reassemble the flow and perform pattern matching. The detection, unlike using BFs, can be completely performed in the fast data path, thus speeding up the overall performance of IDSs.

2.5.3 The anti-evasion system

2.5.3.1 Motivations and Ideas

The main idea of our system is to split a priori the strings to be searched in 3-byte long substrings and create a CBF representing them (hereafter called “substring CBF” or

simply SubCBF) for a preliminary pattern matching. When a substring is detected through the SubCBF, a bank of further filters (called “string CBFs”, StriCBFs) is properly set for the specific flow: more precisely, a filter is initialized for each string which the detected substring belongs to. In this way, all the next packets of that flow are processed in search of the remaining characters of the strings: whenever a StriCBF is completely reset to zero, the attack is detected and the flow is blocked.
However, not all the attacks can be detected in this way; for instance, a string split in several very small packets (less than 3 bytes) is not revealed, because the substring detectors search only for substrings of 3 bytes. Therefore we plan to divert such packets (very infrequent in real traffic) to the slow data path, for flow reassembly and pattern matching. Moreover we set a threshold on the maximum number of flow to be diverted, thus avoiding denial of service attacks.
The only assumption we make is that packets entering our anti-evasion system are not ambiguous, that is packets do not overlap. To force this condition, we assume to have a traffic normalizer (like the one described in [53]) before our system, which keeps traffic flows consistent and solves any ambiguities. Instead, the arrival of out-of-order packets does not affect the correct functioning of the system and the proper detection of attacks.

2.5.3.2 System Architecture

Our architecture, shown in figure 2.24, is composed of several modules. At first, traffic flows are divided by a classifier according to transport protocols, and forwarded to different engines, named substring detectors (SDs). Such a first division allows to balance the load among the SDs and decrease the size of their relative filters. Each SD performs a pattern matching on the overall content of packets by using its specific SubCBF, which represents the set of strings that identifies the valid attacks for that protocol. More precisely, a SubCBF represents substrings of 3 bytes; such a specific length has been selected to reveal also the shortest strings, which are 6-byte long, as pointed out from the analysis of SNORT data sets.
Consequently, a substring detector processes all the ingoing traffic of a specific transport protocol, by moving along an inspection window of 3 bytes. When all the hash functions applied to a group of 3 bytes point to full bins of the SubCBF, the SD determines that the substring has been detected. As previously mentioned, the use of CBFs in this phase allows for a fast update of string set. In particular, we choose MultiLayered Compressed Counting Bloom Filters [54] to implement SubCBFs: the first layer, which is used for the frequent lookups, can be placed in a fast and small memory, while the other layers, useful for string set updates, can be stored in a slower memory.
After a malicious substring is detected by one of the SDs, a block of StriCBFs are set, in order to determine if such an alert actually corresponds to a complete attack. These filters are built a priori and stored in memory: they are addressed by simple hash tables, which in turn are efficiently indexed by the bins of the SubCBFs (as suggested in [55]). The StriCBFs are then handled by other modules, the so-called pattern matching engines (PMEs), which take care of processing the specific traffic flow that the classification stage forwards to it (by means of a classification rule set by the SD after the substring detection). The target of each PME is to perform an overall pattern matching on the flow to determine if the detected substring is actually a piece

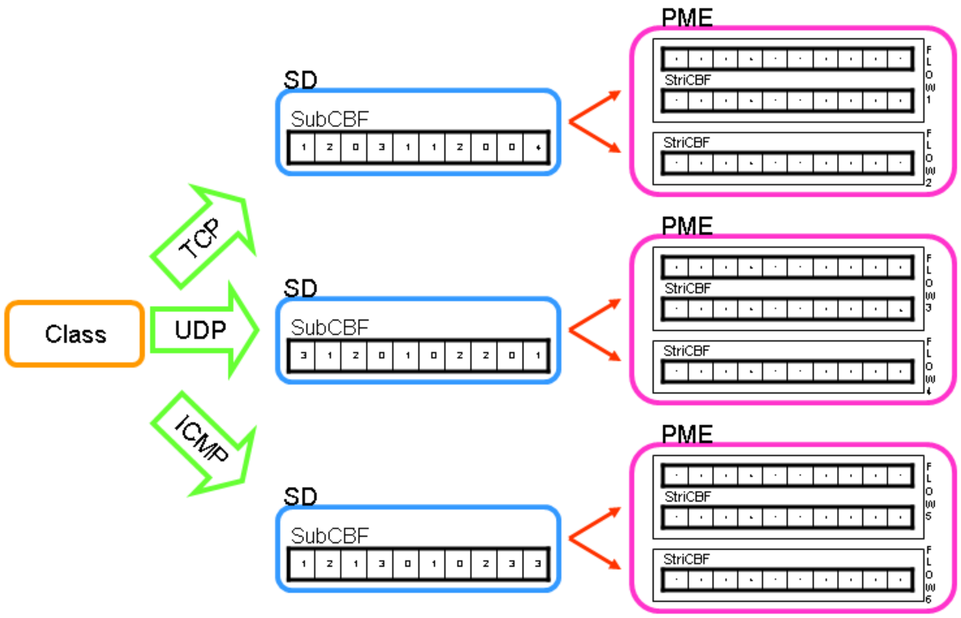


Figure 2.24: The scheme of our system.

of a string or simply a false positive. For this purpose, the PME sets a StriCBF for each string which the detected substring belongs to. Such a CBF represents all the remaining characters of the string in the format $(char, pos)$, where pos is, for example, the TCP sequence number of the character $char$. More precisely, the sequence number of the byte which gets off the string is associated to the filter, so pos is actually the relative position with respect to such a value. Whenever a StriCBF is completely reset to zero, it is assumed that the string has been detected and the packets must be dropped in order to nullify the attack.

Furthermore, also the string length is associated to the filter; from such a value and from the beginning point of the string, the engine is able to understand which bytes of the flow it must process and which bytes cannot belong to the string. In addition, by processing the proper fields of the TCP header, the engine can determine when the filter must be removed because it "has expired".

Clearly, the bins corresponding to the characters of the detected substring which has started the filter are decremented in the initialization phase. In this way, the functioning of PMEs is independent of the arrival order of packets: also a "middle" substring which arrives as first leads off the StriCBF setting and this does not affect the correct functioning of the system.

2.5.3.3 Small packets

If an attacker splits the signature in several 1 or 2-byte long packets (hereafter we call them "small packets"), the system is not able to detect the attack, because the

substring detectors search for substrings of 3 bytes. Fortunately, packets of 1-2 bytes are very rare in real traffic[1], except for certain application such as telnet and ssh, therefore we can use their presence as an alert and adopt proper expedients.
In previous work on anti-evasion systems, this type of attack has been only partially faced. In [51], a Prefix Bloom Filter is set for each substring length, but filters representing pieces of 1 or 2 bytes require an excessive amount of memory; moreover they trigger a hard processing for each packet and an intolerable alert rate (each time a character belonging to a string is found, an alert is generated). Instead, the authors of [52] propose a heuristic very hard to be applied in practice, as illustrated in section 2.5.1.
Our idea to thwart such type of attacks is to divert all the small packets to a slow path engine; this policy is based on the consideration that they are very infrequent in real traffic. The slow path engine has to reassemble such flows and perform a deterministic pattern matching on them, to verify the actual presence of an attack. In order to face denial of service attacks, we also select a threshold on the maximum number of flows to be diverted. Otherwise, attackers could inject a small packet for each flow and force the system to divert and reassemble all the flows.

2.5.4 System Optimization

Our system can be improved, in terms of both functioning and performance, by adopting a series of refinements.
Analyzing real data sets of malicious signatures, we noticed the presence of a few substrings which are very frequent, both in the malicious strings and in normal traffic. We plan to delete such substrings from the filters, thus saving memory (smaller filters) and processing load (less substring which generate an alert). What are the potential drawbacks? Deleting such frequent strings from the SubCBFs could result in a lower detection capability, since less substrings signal an attack. Furthermore, the deletion from the StriCBFs could increase false positives, since filters can be more easily reset to zero. However, the experimental results in section 2.5.5 show just a slight increase of detection capability and decrease of false positives, hence the adoption of such an expedient turns out to be convenient.
Our standard system does not detect all possible patterns of attacks. For example, in some cases the beginning of a signature is missed, because it is too short to be revealed by the substring detectors, while it belongs to a packet which is too long to be classified as "small" (as *SI* in figure 2.25). Therefore the packet is identified as normal, and whenever another fragment (*GNATU* in figure) triggers a StriCBF setting, this piece has already been processed. In this way, some bins will continue to be full and the filter will never be completely reset to zero. Moreover, in order to speed up processing, one might consider not waiting for the overall emptying of filters. Therefore, for the efficiency of our system, it seems advisable to set an "emptying threshold" α for the StriCBFs: when α is exceeded, the attack is considered as detected. Such a threshold is computed as the ratio between the number of bin decrements and the sum of all the bin counters, where α equal to 1 means that the overall filter has to be depleted, while lowering the threshold results in a faster detection. It is necessary to find the correct

[1] as shown for example by data at http://netflow.internet2.edu/.

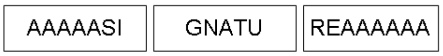


Figure 2.25: The string is SIGNATURE. The piece SI is not outright detected, and when the piece GNA generates an alert, it has already been processed.

trade-off between a higher speed and a larger number of potential false positives (as shown in section 2.5.5).
With respect to the memory efficiency, while CBFs (or their improved versions ML-CCBF [54], Blooming Trees [56] or dl-CBF [57]) are the best choices for SubCBFs, StriCBFs may not require the minimum amount of memory for their function, which is to recognize simple characters. Thus, StriCBFs can be replaced by the actual signature string and a bitmap (1 bit per signature byte) that indicates whether each character has been found or not; by pursuing this approach, we use 9 bits per signature byte. Instead, with a plain CBF, we would need a number of bits per character equal to $4k/\ln 2$ ($k = \log_2 f$ where f is the false positive probability), which falls down approximately to $k/\ln 2$ bits if a Blooming Tree (BT) is used. Of course when k is large, this amount can be larger than 9 bits.
Therefore, the choice of either using the string itself or a CBF (or, better, a BT) is related to the parameter f, which is, in turn, computed according to the total number of times n_q we query that particular filter. In fact, the mean number of false positives is given by the product of f and n_q; as we do not have overlapping packets (because of the normalizer), the number of times n_q we check a given StriCBF that represents a n-bytes string is exactly n. Then, by simply selecting $f \times n = 2^{-k} \times n \ll 1$, we can cope with false positives by largely limiting their mean number. In practice, we have found that when $n < 10$ (remember that StriCBFs do not include the 3-byte substrings found by SubCBFs, thus further reducing n), we can achieve to use less than 9 bits per character without increasing the amount of false positives. In the experiments shown in section 2.5.5, we used the combinations of BTs and strings+bitmaps that minimize the amount of memory used.

2.5.5 Experimental Results

For the experimental runs, a cluster of PCs which generate traffic towards a LAN is used: one of them runs FTester, which is a software tool designed for testing IDSs capabilities, while the other ones generate background traffic. A general purpose PC running our anti-evasion system is placed before the LAN to protect it. We do not need to use a normalizer (however included by our system), because we set FTester to generate attacks with no ambiguities.
FTester is able to create evasion attacks, by splitting signatures among several packets and with different lengths and number of signatures, order of pieces, and so on. In particular, we use the following options of FTester (let us suppose the malicious string is ATTACK):

- *-e stream* (simple splitting of the tcp stream): packet = [packet1(ATT) + packet2(ACK)]

- *-e frag1* (out of order packets): packet = [fragment3(C) + fragment2(TA) + fragment1(AT) + fragment4(K)]
- *-e frag2* (like frag1 but send the last fragment first): packet = [fragment4(K) + fragment3(C) + fragment2(TA) + fragment1(AT)]

The lengths of substrings in our runs are alternated in order to have both "normal" evasion attacks (with substrings of almost 3 bytes) and attacks with "small packets" (less than 2 bytes).

Table 2.9: Performance of the standard system in terms of detected attacks and false positives.

Trace	Size (MB)	Normal attacks			Small attacks		
		gen.	detect.	false pos.	gen.	detect.	false pos.
Tr1	224	4240	99.8%	0.23%	312	100%	4.8%
Tr2	190	3800	98.9%	0.1%	310	100%	4.8%
Tr3	160	1418	99.9%	0.35%	200	100%	3.5%
Tr4	100	1213	100%	0.32%	185	100%	4.3%
Tr5	50	789	99.2%	0.25%	108	100%	6.4%

In table 2.9, for traces of different sizes and by distinguishing between "normal" and "small" attacks, the following data are reported:

- The number of attacks which are generated.
- The percentage of real attacks we detect.
- The percentage of false positives.

The last three columns enumerate the total values. Such measurements refer to the standard functioning of the system; the effects of the optimizations listed in section 2.5.4 are shown later.
These results exhibit high percentages of detection, while the number of false positives remains small. As foreseen, the technique used against the attacks performed with "small packets" generates the largest number of false alerts, since each small packet is signaled as potential attack.
In table 2.10, the potential drawbacks of deleting off-line the most frequent substrings from the filters are shown. The traces under test are the same of table 2.9 and we refer only to the "normal attacks" (i.e., column 3-5 in table 2.9), since the "small attacks" are not affected by such a modification. The number of detected attacks by deleting such substrings ("No Frequent Substrings", NFS, in the table) is practically equal to those revealed by the standard system (ST), while the number of false positives slightly increases. These results justify the adoption of such a refinement, which, without remarkable additional costs, improves the efficiency of the system in terms of both memory and speed. Specifically, by adopting such an expedient, we observe a mean reduction of memory footprint by a factor of 4 in the experiments, thus requiring less than 100 bytes for each flow processed by the PMEs.

Table 2.10: The effects of deleting the most frequent substrings.

Trace	Detected att. (%)		False pos. (%)	
	NFS	ST	NFS	ST
Tr1	99.7	99.8	0.26	0.23
Tr2	98.9	98.9	0.13	0.1
Tr3	99.8	99.8	0.35	0.35
Tr4	99.8	100	0.41	0.32
Tr5	99.2	99.2	0.5	0.25

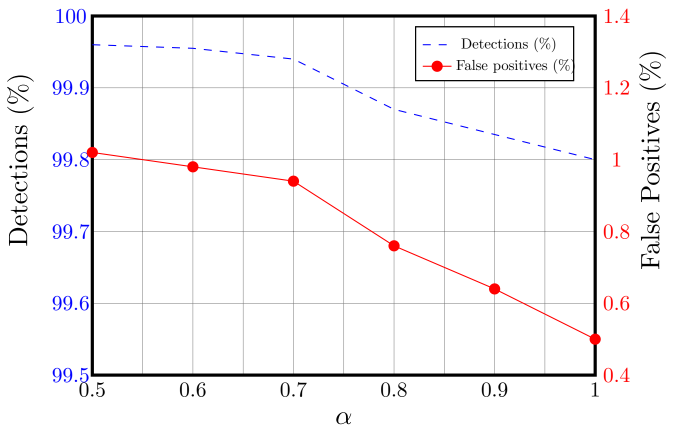


Figure 2.26: Detection percentage and false positives by varying α.

Finally, figure 2.26 shows, for the processing of the trace Tr1, the effects of the "emptying threshold" α, which allows us to detect a string even though the relative StriCBF has not been completely reset to zero. It is clear that choosing low values of α results in a higher percentage of detection but, also, in a larger number of false positives, while, for high values, the opposite happens. In any case, the number of detected attacks is sufficiently high (i.e., beyond 99%).

Chapter 3

Perfect Hashing Schemes for Data Indexing

Hash tables are used in many networking applications, such as lookup and packet classification. But the issue of collisions resolution makes their use slow and not suitable for fast operations. Therefore, perfect hash functions have been introduced to make the hashing mechanism more efficient. In particular, a minimal perfect hash function is a function that maps a set of n keys into a set of n integer numbers without collisions. This chapter illustrates two perfect hashing schemes, which are useful in devices provided by a limited amount of memory, such as Network Processors or FPGA-based networking boards. Memory saving, which is a paramount issue in networking applications in hardware, lead us towards the design of different schemes and algorithms with many appealing properties. All these solutions are based on principles of Bloom Filters, which are efficient randomized data structures for membership queries on a set with a certain known false positive probability. A brief introduction to Hash Functions and Bloom Filters could be found in appendix A.1 and in appendix A.2.

3.1 Minimal Perfect Hashing through Bloom Filters

Hash tables are frequently used in networking applications. They can be found in compilers, language translation systems, and information retrieval. But the issue of collisions resolution makes their use slow and not proper for fast operations. Therefore, perfect hash functions have been introduced to alleviate these limitations and to improve performance. A perfect hash function maps a static set of n keys into a set of m integers without collisions, where m is greater than or equal to n. If m is equal to n, the function is called minimal.

Minimal perfect hash functions (MPHFs) are widely used for memory efficient storage and fast retrieval of items. They can be used also for security purposes: the capability of efficiently revealing the presence of certain strings allows for a fast detection of attacks or for determining which data have to be anonymized in privacy-preserving approaches [58]. Given the high operating speeds of current links, item retrieval must be very fast. Moreover, item sets can be very large (e.g., search engines are nowadays indexing tens of billions of pages), thus these algorithms must be very space-efficient. To summarize, the goodness of a MPHF scheme depends on the time and the space needed for its construction, on the time required by the MPHF for each retrieval of an element and on the number of bits needed to represent the element. While CAM/TCAM hardware is a fast (yet energy-greedy) alternative to implement the same functions, the recent general trend for energy savings and the need for cheap and general implementation contribute to keep MPHFs important and attractive in network devices.

This work presents a new technique to construct a minimal perfect hash function by using specific data structures based on Blooming Trees [56]. The main objectives are:

- simple construction process;
- fast retrievals;
- memory savings;

The target platforms are network processors (NPs) or general purposes processors (GPPs) that provide the "popcount" instruction to compute the number of "1" bits in a word (for instance Intel® Itanium [59], the future Nehalem [60] and the IXP2800 [61], AMD® Phenom [62] and IBM® Power6 [63]).

3.1.1 Related works

In the following, we present the major results about the construction of MPHF. For further details, [64] gives a comprehensive survey on perfect hashing. Fredman, Komlós and Szemerédi [65] present a space efficient structure to construct MPHFs, which uses a space of the order of $n + o(n)$. The construction time of this model, based on hashing properties, is $O(n)$, and the same result is also obtained in [66, 67]. Mehlhorn [68] shows that at least 1.44 bits per key are needed to represent a MPHF. Fox et. al. [69] illustrate an algorithm whose encoding size is very close to such a theoretical bound (i.e., around 2.5 bits per key) and which uses the well-known *mapping-ordering-searching* scheme. However, [64] proves that such a scheme has exponential running times. Pagh [67] proposes a new way of constructing MPHFs through randomized algorithms. The form of the resulting function is $h(x) = f(x) + d[g(x)]mod(n)$, where f and g are hash functions and d is a set of values to resolve collisions. The hash function description occupies $O(n)$ words and can be constructed in $O(n)$ expected time. Czech et al. [66] introduce a new algorithm for MPHF which preserves the elements order. It involves the generation of random graphs; the time complexity is $O(n)$ and the space required to store the function is $O(n \log n)$, which is optimal for order preserving MPHFs [64]. This algorithm takes $32.9s$ to construct a MPHF for 524288 keys on a Sequent machine. Botelho et al. [70] propose a solution based on the classic divide and conquer technique, which is capable of generating MPHFs for sets of billion of keys. The construction time is $O(n \log n)$, the evaluation of $h(x)$ requires 3 memory accesses for any key x and the description of h takes a number of up to 9 bits for each key, which is optimal for huge sets. To the best of our knowledge, the solution which offers the best tradeoff between construction time and storage space is illustrated in [71]. It uses r-uniform random hypergraphs given by function values of r hash functions on the keys to be processed. Such an algorithm will be the reference for the performance evaluation of our solution. Finally, [72] introduces a novel scheme for MPHF which requires about 8.6 bits per key. The construction is several orders of magnitude faster than existing perfect hashing schemes based on mapping-partitioning-searching model, because searching is avoided. Bloom Filters (BFs) are employed, which are known for simplicity and speed. This schemes, running on a Pentium IV, needs $7.73s$ to construct a MPHF for 3.8 millions of keys and $125ms$ for 110 thousands of keys. It inspires this work in the use of BF-like structures for MPHFs. Instead of standard BFs, a composed data structure is used: a first level is given by a Huffman Spectral Bloom Filter (HSBF) [54] while the remaining part is based on a novel filter, the so–called Blooming Tree (BT) [56]. This way, a novel method is proposed, which allows for an easy MPHF construction and fast retrieval, with low memory requirements.

3.1.2 What is a Blooming Tree?

The idea of Blooming Trees [56] is constructing a binary tree upon each element of a plain BF, thus creating a multilayered structure where each layer represents a different depth-level of tree nodes. The aim is to achieve both low false positive probability and low memory requirements, while the drawback is the increased cost in lookup operation. The latter can be mitigated by the low memory consumption, that enables the deployment of the structure in faster on-chip memories.
To build a Naive Blooming Tree (NBT) for n elements, $L + 2$ layers are defined:

- a plain BF (B_0) with k_0 hash functions h_j ($j = 1 \dots k_0$) and m bins such that $m = nk_0/\ln 2$;
- L layers ($B_1 \dots B_L$), each composed of m_i ($i = 1 \dots L$) blocks of 2^b bits;
- a final layer (B_{L+1}) composed of c-bits counters.

The j-th hash function h_j provides a $\log_2 m + L \times b$ bit long output: the first group ($s_{0,j}$) of $\log_2 m$ bits is used to address the BF at layer 0, the other $L \times b$ bits are divided into L substrings ($s_{1,j} \dots s_{L,j}$) of b bits, one for each layer. The lookup for an element σ consists of a check on k_0 elements in the BF (layer 0) and an exploration of the corresponding k_0 "branches" of the Blooming Tree. The overall process of lookup is accurately explained in [56]. An optimized version of BT [56] follows from some observations about NBTs. In particular, the "zero-blocks" are used to stop the "branch" from growing as soon as the absence of a collision is detected in a layer (which entails for construction the absence of collisions in all the upper layer of the branch). However, to keep construction and lookup possible, the Optimized BT (OBT) employs a bitmap and an array of hash substrings for each layer. The array of substrings for a certain layer is composed of all the hash substrings that complete the hash of the "branches" that stop at that layer, while the bitmap addresses such substring array.

3.1.3 The MPHF construction

Our algorithm is based on the statement that, given an ordering algorithm g and a set S, a MPHF of an element $x \in S$ is simply the position of x in the given ordering scheme:

$$\text{MPHF}(x) = \underset{S,g}{\text{position}}(x) \tag{3.1}$$

3.1.3.1 Using the Naive Blooming Trees

The basic idea of our algorithm is to use a BT as ordering algorithm for the set of elements S we have:

$$\text{MPHF}(x) = \underset{S,BT}{\text{position}}(x) \tag{3.2}$$

In particular, the NBT can be used for this purpose with no modifications. The construction is exactly the same as the one introduced in section 3.1.2, using a single hash function. All we need to care is to make sure that the counters at layer $L + 1$ are all equal to 1, which means that all the elements have been separated. In this situation, in order to evaluate MPHF(x), a single lookup operation is required: once the corresponding (say, the j-th) counter of x in layer B_{L+1} is found, we return j (obtained through a simple popcount) as the result. We need to design the structure to have very low probability of collisions in the last layer (B_{L+1}). This, in turn, results in a high probability of obtaining a successful MPHF construction in a few attempts. If the construction is successful, we achieve our ordering scheme: the element that falls into the first counter is hashed to 0, the element falling into the second one is hashed to 1, and so on. Therefore, our hash function is perfect and also minimal, in that we assign the first n integers as the results of hash retrieval for n elements.

3.1.3.2 Using the Optimized Blooming Tree and the HSBF

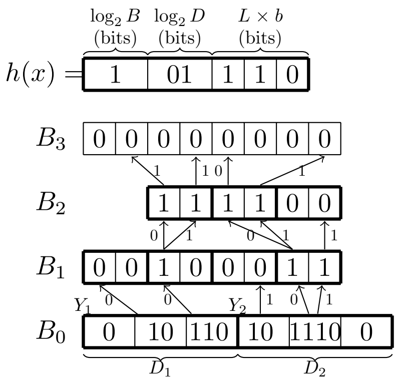
The OBT is an elaborated version of the NBT that improves memory efficiency, thus being attractive for our purposes. The idea is blocking branch from growing, as soon as an element does not experience any collision, by using the zero-blocks as leaves of the trees. However, this expedient removes the last layer, which till now provided a simple way to compute a MPHF by means of popcounts. Recall that the problem lies in how to compute the number of elements at the left of a given element x. Our idea to solve this problem is to divide the procedure into two steps:

- find the tree which x belongs to (we shall call it T_x) and compute the number of elements at the T_x's left;
- compute the number of leaves at the left of x within T_x.

In order to do so, we propose the HSBF [54] as the first level of the BT, instead of the standard BF [56]. The HSBF is composed of a series of bins encoded by Huffman coding so that a value j translates into j "1s" and a trailing "0". Therefore, the first step (i.e.: computing the number of elements in the trees at T_x's left) is obtained by a popcount in the HSBF of all the bins at the left of x's bin. As for the second step, we have to explore (from left to right) the tree T_x until we find x, thus obtaining its position within the tree. The sum of these two components gives the hash value to be assigned:

$$\text{MPHF}(x) = \text{popcount}(\text{HSBF}[x]) + \underset{T_x}{\text{position}}(x) \tag{3.3}$$

Notice that, in a standard BT, the popcount in the first layer gives the block to be read at the next layer. To achieve the same functionality in an HSBF, it suffices to count the number of bins greater than 0 (i.e.: the number of "10" bit-sequences). The HSBF is divided into B sections of D bins, which are addressed through a lookup table. Each entry of this table keeps all the necessary information for a section: the starting address in memory, the number of elements that fall in the previous sections (which are computed by means of popcounts, as stated above), and the number of "10" bit-sequences found in the previous sections. The OBT has a maximum of L layers ($B_1 \dots B_L$), each composed of blocks of 2^b bits.
A hash function $h(\cdot)$ is used. Its output is $\log_2 B + \log_2 D + Lb$ long bits: the first $\log_2 B$ bits indicate the section and address the lookup table, the subsequent $\log_2 D$ bits index the bin within the section in the HSBF, while the last ones are divided into substrings of b bits, one for each BT layer.
A simple example (see fig.3.1) clarifies the procedure. Let us assume $B = 2$, $D = 3$, and $b = 1$: hence, the hash output is 6-bits long. Let us suppose $h(x) = 101110$. The first bit is used to address the lookup table: it points to the second entry. We read the starting address of section D_2 and that 3 elements are in the previous sections. Now we use the next two bits of $h(x)$ to address the proper bin in section D_2: "01" means the second bin. The popcount of the previous bins in the section indicates that another element is present (so far the total of elements at T_x's left is 4). Then we care about T_x: to move up to the next layer, we both use the third information in the table and count the number of "10"s in the previous bins of this section. The sum shows that, before our bin, 3 bins are not equal to 0, so we move to the fourth block in layer B_1.
Here, the fourth bit of $h(x)$ allows to select the bit to be processed: the second one. But we want to know all the T_x's leaves at x's left. Hence, we must explore all the branches starting from the first bit of the block and count the number of zero-blocks we find: it is 2 (now, the counter reads 6). Regarding the second bit, a popcount in layer B_1 indicates the third block in layer B_2: it is a zero-block, so we have found the block representing our element: x is the 7-th element in our ordering scheme. Then $\text{MPHF}(x) = 6$.



Lookup Table

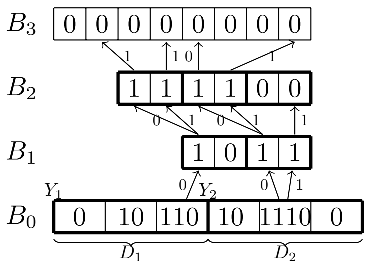
Start addr.	Prev. elements	Prev. "10"s
Y_1	0	0
Y_2	3	2

Figure 3.1: Example of hash retrieval by using OBT and HSBF.

An obvious objection is that a lookup may require many jumps and be expensive since we need to explore, on average, half a tree to find our element. However, this is not a big issue because all the nodes of a tree at the same layer are contiguous in memory and can be accessed (and cached) in a single memory reference. Hence, the number of accesses for an element is simply the depth of the tree it belongs to. Finally notice that, in the evaluation process, bitmaps and hash substrings tables have not been used; therefore, after the MPHF construction, they can be removed from memory.

3.1.3.3 Using a more efficient version

Another potential improvement becomes clear as we make the following remark on the above described structure: whenever a bin in the HSBF is equal to 1 (i.e.: it reads "10"), it is a waste of memory to allocate its zero-block at the next layer, because only a single element falls into it. Since the probability of having a bin equal to 1 is larger than the probability of large bin values in a CBF (see eq. (3.4) in the next section), this improvement notably reduces the average cost in terms of lookup time and memory size. The construction process does not change but, after the construction, the structure can be reduced according to the previous discussion. Also, the lookup table must be modified: the third element of each entry must now indicates the number of "110" bit-sequences in the previous sections of the HSBF, because only bins strictly greater than 1 have a corresponding block at layer 1 under this new scheme. The example in fig. 3.2 shows the reduction of the structure of fig. 3.1: we observe the deletion of the first and the third blocks in B_1, which were related to the second and the fourth bins in the HSBF (the "10" bins), and the change of the lookup table.



Lookup Table

Start addr.	Prev. elements	Prev. "110"s
Y_1	0	0
Y_2	3	1

Figure 3.2: Example of hash retrieval in the optimized structure.

3.1.4 Complexity and properties

In order to simplify the rest of the analysis, it is useful to remind one of the main results of [56]:

$$P_i(\varphi) \simeq \frac{e^{-\alpha_i}\alpha_i^{\varphi}}{\varphi!} = \text{Poisson}(\alpha_i, \varphi) \quad \text{with } \alpha_i = 2^{-i}\ln 2 \tag{3.4}$$

Eq. (3.4) claims that the number of elements φ colliding in any block of layer i can be well-approximated by a Poisson pmf with parameter α_i. This result provides a tool to design our MPHF. In particular, we can compute the number of layers needed to guarantee a fast construction ("fast", here, means "within one or few trials") of our BT by simply setting:

$$n \times P_{L+1}(2) \simeq q \tag{3.5}$$

where $q \leq 1$ is the probability of having at least a bin greater than 1 at layer $L+1$; in addition, q corresponds to the probability of the unlucky event that we need to retry the construction (it is also called the failure probability [72]).
In that event, a different approach can be pursued by just adding extra layers until the collisions disappear. This requires, of course, the output of the hash function $h(\cdot)$ to be wider than $\log_2 B + \log_2 D + Lb$ bits, but it can be less expensive than restarting the entire construction from scratch.

3.1.4.1 Memory requirements

The average size of our MPHF can be computed as the sum of its components. As for the OBT, since we do not need all its complementary structures such as bitmaps and hash substrings, we compute its size as the sum of leaves and non-leaves nodes. In the structure described in section 3.1.3.2 the number of leaves is simply the number of elements n. However, in the optimized structure described in section 3.1.3.3, we delete the layer 1 leaves, thus obtaining $n - m_0P_0(1) = n(1 - P_0(1)/\ln 2)$ leaves. On the other hand, the number of non-leaves nodes can be computed as $m_0P_i(\varphi > 1)$.

Table 3.1: Memory requirements in bits/key.

b	W_{HSBF}	W_{OBT}	ΔS_{HSBF}	ΔS_{OBT}	S'	S_{tot}
1	512	512	0.45	0.10	3.63	4.18
1	1024	1024	0.23	0.06	3.63	3.92
1	2048	2048	0.11	0.03	3.63	3.77
2	512	512	0.45	0.10	4.82	5.37
2	1024	1024	0.23	0.06	4.82	5.11
2	2048	2048	0.11	0.03	4.82	4.96

Thus the average memory size of our OBT is:

$$S_{OBT} = 2^b \left(n(1 - P_0(1)/\ln 2) + m_0 \sum_{i=1}^{N_l} P_i(\varphi > 1) \right) \tag{3.6}$$

where N_l is the number of required layers and $P_i(\varphi > 1) = 1 - P_i(0) - P_i(1)$ can be computed by means of (3.4).

When dealing with perfect hash functions, it is common to express the memory requirements in terms of bits per key:

$$S/n = 2^b \left(1/2 + \frac{1}{\ln 2} \sum_{i=1}^{N_l} P_i(\varphi > 1) \right) + 1 + \frac{1}{\ln 2} \tag{3.7}$$

A first comment is that $b = 1$ is the less expensive choice in terms of memory consumption. Larger values of b reduce the depth of the blooming tree (that is the number of non-leaves nodes) but its contribution to S/n in (3.7) is negligible. Therefore, $b = 1$ shall be the preferred setting hereafter. Moreover, we notice that the optimization process discussed in 3.1.3.3 reduces the number of bits-per-key metric of 0.5 bits. However a number of tables are needed in order to make the lookup phase faster and more manageable. In fact, both the HSBF and the upper layers can be divided in sections so that accessing them or computing a popcount requires less time. We already discussed in section 3.1.3.2 about the tables for the HSBF. Now, we also consider dividing each layer of the OBT into sections and adding, for each layer, a table whose j-th entry reports the popcount of all bits before section j.

Naturally, the increment in memory requirements introduced by these tables depends on B and D. If we focus on the bits/key metric, we can compute the table cost in memory through b and W only (i.e., the bit size of sections). Tab. 3.1 reports the consumption in bits/key for the total structure (S_{tot}) along with the contributions of lookup tables for the HSBF (ΔS_{HSBF}) and the OBT (ΔS_{OBT}), which are added to the main structure S'. A good choice is $W = 1024$ bits: all the tables cost only 0.29 bits per key, thus bringing the final memory requirement to 3.92 bits per key. Moreover, in modern 64-bits processors, 1024 bits represent 16 words only, and can be read in a single memory access. However, other values of W do not significantly change the final memory budget.

Table 3.2: Algorithm comparison.

Algorithm		Evaluation		Construction Time		bits/key
		time(s)	mem.ref.	mean(s)	std.dev	
Our	$m = 2^{22}$	1.21	3.1	12.78	0.11	4.02
	$m = 2^{23}$	0.98	2.53	13.37	0.14	4.75
BPZ		1.35	-	18.37	4.41	3.60
BL		-	2.38	7.73	-	9.1

3.1.4.2 Hash evaluation cost

In the following study on the average cost of a lookup, we focus on the number of memory accesses. Indeed a memory access commonly requires hundreds of clock cycles, thus accounting for almost the totality of the hash evaluation cost.
During a lookup, we have to compute a hash function and use its output to address the lookup table and the HSBF. Moreover, if the bin we read reports a collision (i.e.: more than 1 element falls into it), we need to explore a certain number of layers according to the depth of the resulting tree. Eq. (3.4) comes in handy also in this computation. It expresses the pmf of m bins, but we do not care about empty bins. Therefore we need to normalize the pmf in (3.4) by dividing it by $(1 - P_0(0)) = 1/2$:

$$P_i'(\varphi) = \begin{cases} \frac{P_i(\varphi)}{1-P_0(0)} = 2 \times P_i(\varphi) & \varphi \geq 1 \\ 0 & \varphi = 0 \end{cases} \tag{3.8}$$

Of course, $P_i'(0) = 0$ because we will not lookup empty bins. Then $n \times P_0'(1)$ times we will access only the lookup table and the HSBF. This costs two memory accesses (if a HSBF section is read in a single access). In all other cases ($n \times (1 - P_0'(1))$ times), we have a tree to explore.
As previously mentioned, in our construction all nodes of a tree at the same layer are contiguous in memory. This means that, as a matter of fact, when accessing a given node we read (and cache) all the other nodes at the same layer. Therefore the average number of memory accesses for the tree exploration is simply the average tree depth $\overline{d} = \sum_{i=1}^{L+1} i \times P_i'(1) \simeq 2.4$. Finally, the average number of memory accesses is:

$$\overline{w} = 2 + (1 - P_0'(1))\overline{d} \simeq 2.73 \tag{3.9}$$

It does not depend in any way on the number of elements, hence the lookup cost is $O(1)$.

3.1.5 Experimental Results

We compare our algorithm to the one proposed by Bonomi and Lu [72] (hereafter called BL) and to the fastest and least memory-requiring algorithm that we found [71] (BPZ). We are aware that, because of the processor evolution and the limited availability of the code of other algorithms, it is always difficult to present fair comparisons for algorithms. We tested an implementation of our algorithm (developed in

C) on an Intel 2.4 Ghz Pentium 4 Core 2 Duo processor (4 MB L2-Cache), equipped with 4 GB of RAM and running Linux OS 2.6, while BPZ employed a 3.2 Ghz XEON (2 MB L2-Cache), with 1 GB of RAM running Linux 2.6 and BL describes its test platform simply as Pentium 4. Even if the processor we used is dual-core, this does not give us any advantage because our implementation is sequential and runs on a single core. This means also that, even if the L2-cache is 4 MB, only half of it is available (on average) to our algorithm. Both papers (on BPZ and BL) present their results for a similar number of keys (3.541.615 for BPZ and 3.8 million for BL) thus we set $n = 3.8 \times 10^6$ in our algorithm.
In tab. 3.2 we show a comparison of BPZ, BL and our algorithm in terms of construction and lookup time, as well as memory requirements. The lookup time is obtained by querying the MPHF for all the 3.8×10^6 keys in random order.
Since in our algorithm we would like to have $m \simeq n/\ln 2$, we have two choices for the first layer: $m = 2^{22}$ or $m = 2^{23}$. We tested both. In the first case, we measured 3.1 memory accesses (on average) and 4.02 bits per key, while in the second case we had a faster lookup, but an increment of about 0.7 bits per key in the size and of more than 0.5s in the construction time. Such values confirm the theoretical results of the previous section. As for the failure probability, by limiting the number of layers to 10, it turned out to be less than 5×10^{-4} in all cases. However, on processors such as the ones cited in sec. 3.1, both construction and lookup times will decrease because of the high frequency of popcount calls. Results clearly show an improvement in terms of construction and lookup time as compared to BPZ, at the cost of a slight increase in memory requirement. Instead, compared to BL, our solution halves the bits-per-key metrics, while slowing down lookup and construction processes.

3.2 iBF: Indexed Bloom Filter

Since BFs are randomized structures that rely on hash functions, they allow a certain amount of false positives; however, the advantage of the space savings often outweighs such a drawback.
In order to minimize false positives, the parameter k can be set according to a well-known result: $k = m/n \ln 2$, where n is the number of elements represented in the BF. In such a condition, false positive probability reaches the minimum value of $f = 2^{-k}$.
Although BFs have many features that make them attractive for fast and simple applications, their adoption in more sophisticated schemes is prevented by their lack of functionalities and (in some cases) poor performance. As a motivational example, let us suppose we need to classify traffic according to fixed-size substrings of packet payloads at high speed. Let us assume that we are looking for a set of particular strings. We can train a BF with the set of pre-determined strings we are searching, so that the (hopefully large) part of traffic that does not match them can pass through with no additional computation required. However, if the BF returns a match, we need then to check whether it is a false positive and which string has been matched. This requires an additional *exact* filtering stage, which could be implemented as an hash-table. Moreover, while the first stage may help the second hash-table lookup (as, for instance, shown in [55] where a Counting Bloom Filter reduces the number of accesses to the following hash table), the whole lookup remains non-deterministic thus jeopardizing performance if implemented in parallel systems. In order to achieve

Term	Description
singleton bit j	CBF[j]=1
marked bit	a singleton bit cleared to 0. One per element.
index(x)	b bits at the marked bit's left
good BF	BF where each element has at least an singleton bit
well-constructed BF	BF with minimum false positive probability

Table 3.3: Terms and notation used through the work

deterministic lookup times, a perfect hashing scheme ,as shown by Kumar et al. in ([73][33][74]), is very effective. These works propose the adoption of a small fast table of "discriminator" values which, together with the key, are fed to a regular hash function thus removing collisions and achieving perfect hashing. In such a way, in [73] and [33], finite automata are succintly stored and in [74] perfect hashing is achieved with as low as 1 additional bit per key in a double hashing scheme. A BF-like structure is also adopted in the previous section where a Blooming Tree is the basis for the construction of a minimal perfect hashing scheme. However, all these results come at the cost of a quite expensive construction and, unlike the iBF, cannot be adopted in existing applications with minimal effort, as they require major code rewriting of even hardware modifications in order to be effective. Indeed, there solutions require more than a single memory block to be effective as they rely on a number of tables to be accessed at the same time. This implies that, in an existing application, more than just code rewriting is needed: new fast memory blocks and corresponding bus bandwidth must be allocated. The purpose of this work is to show we can use a BF to obtain a perfect hashing scheme by exploiting a certain number of degrees of freedom and relaxing the false probability requirements. In details, we show a quite succinct data-structure, which is a direct modification of a BF that can be implemented in existing applications adopting BF at a negligible cost in terms of code rewriting. The modified BF we construct returns an index for each element of the working set, hence the name *indexed BF* or iBF. The data structure requires $O(\log(n))$ bits per key and k memory accesses per lookup, where k is $O(\log(n))$. A closely related work is the one by Chazelle et al. in [75], introducing Bloomier Filters. Bloomier Filters augment Bloom Filters with the capability of storing any function of the input set. They are therefore more general than our iBF but may require a larger amount of memory.
In short, the main contribution of this work is its novel approach, which exploits a couple of interesting degrees of freedom in BFs, to a widely discussed problem: achieving deterministic perfect hashing in network applications. We believe such degrees of freedom may also be useful for other purposes and the algorithm we propose can be adopted with minor changes in existing applications based on BFs.

3.2.1 The main idea

The purpose of iBF is to create a perfect hash by simple bit-flipping operations on an already-constructed BF.
As a motivational example, the structure in fig.3.3 shows our desired result: a BF and 2 elements (x and y) are depicted. For each element, one of the 3 hash functions

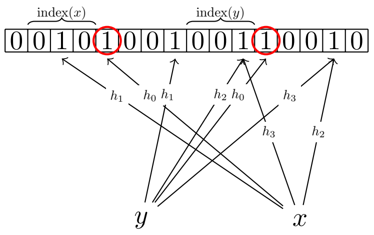


Figure 3.3: The desired data structure

points to a marked bit which, in turn, defines an index at its left. These indexes serve as perfect hash for elements x and y.
The idea of iBF builds upon the following considerations:

1. In a well-constructed BF, if k is large enough (we will show that it must not be larger than $O(\log(n))$) there is at least a hash-function h_i for each item x of the set S that addresses a bit with no collisions (i.e.: where no other $h_j(y)$ falls, $\forall y \in S, y \neq x$). We hereafter refer to BFs with such property as "good" BFs.

2. Bits equal to "zeros" in the BF can be flipped to 1 by only paying a small price in terms of false positives.

These observations basically lead the construction which is, in turn, performed in two steps. The first step marks a bit for each item in the set by focusing on non-colliding bits as suggested by the first observation. The other step exploits the second consideration and flips a number of bits at the left of each marked bit in order to obtain, for each element, a different return value. In the following we describe these operations in greater details.

3.2.2 iBF Construction

3.2.2.1 First step: determine bits to mark

We want to have an univocal index to be returned from the BF for each element x in the set S. To this aim, we take advantage of the first consideration and focus on the "non-colliding" bits in the BF (i.e., bits which have been set by a single element only). In the following we refer to those bits simply as "singleton" (see legend in tab.3.3). A simple way to describe such property is that if we expand the BF to a Counting BF (CBF), singleton bits are those corresponding to counters equal to 1. The first degree of freedom we exploit in this work is used here. By definition, in a good BF, for each element we have at least one of the singleton bits that we can flip to zero, thus marking it. This way, we relax the BF requirements, accepting that an element x belongs to the set if the k hash functions point to k-1 ones and 1 zero. Hence, the false positive probability grows by a factor of 2 (as if we were using a BF with k-1 hash functions), but we earn a way to "mark", for each element x, one of the bits representing it. This is crucial in order to proceed in our construction.
Let us now discuss about the likelihood of the first consideration; in other terms, how probable are good BFs? And what choice of parameters m and k makes a BF good?

The probability for an element x to have at least a singleton bit is simply:

$$\pi = 1 - \left(1 - e^{-\alpha}\right)^k$$

where k is the number of hash functions and α is defined as nk/m. Then, the probability that this property holds for all the n elements (i.e., the probability of a good BF) is:

$$P = \pi^n \simeq e^{-n(1-e^{-\alpha})^k} \tag{3.10}$$

It can be easily demonstrated that the value of k which maximizes P is the same that minimizes false positives: $k = m/n \ln 2$. The reason is simple: let us assume we have $n-1$ items stored in our BF and we add the n-th item. Computing the probability that all the k hash functions point to already-set bits is basically computing the probability of a false positive f. Since we try to avoid this event, maximizing P is the same as minimizing f.

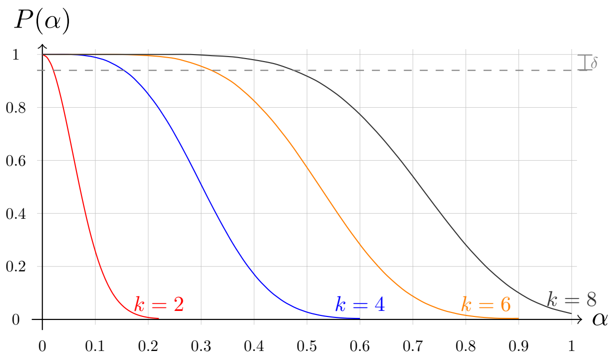


Figure 3.4: Probability of good BFs as a function of α and k.

Naturally, we are interested in making P as close to 1 as possible. In this sense, for $\delta \to 0$, we observe that $P \geq 1 - \delta$ if:

$$k \geq \frac{\log n - \log \delta}{-\log(1 - e^{-\alpha})}$$

The main comment is that k must grow like $\log n$ which is quite intuitive, as it makes the structure size behave as Bloomier Filters: $O(n \log n)$. The effect of this inequality is shown in fig. 3.4 where P is reported for different k and as a function of $\alpha = nk/\ln 2$.

Once we assessed the conditions that make well-constructed BFs probable, we can proceed to determine which singleton bit to mark among the ones belonging to each item. The choice may be driven according to different metrics which can be combined in order to facilitate the second step. In our experimental tests, we found that a good metric is, for a singleton bit j, the number of zeros at j's left minus the minimum distance between j and other singleton marked bits.

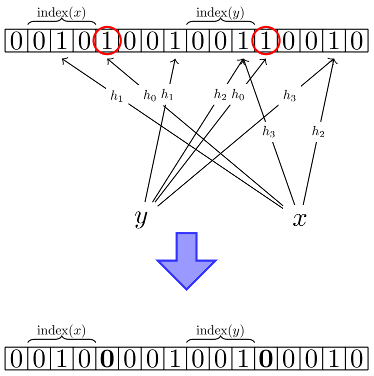


Figure 3.5: Overall scheme. Here the parameters $\varepsilon = 2$ and $m = 16$ are quite over-dimensioned in order to better illustrate the idea.

3.2.2.2 Second step: build the index

As a second step, we need to get an index out of the BF for each element $x \in S$. In order to do that, we use the marked bits, and simply choose the index to be defined by a number of $b = \log_2 n + \varepsilon$ bits at the left[1] of them (as shown in fig.3.5 where marked bits are circled). In the following, we will refer to those b bits simply as "indexes". As we have an index for each of the n elements of the set, we can then determine them referring to their corresponding element: for instance, index(x) refers to the b bits at the left of the marked bit for x.

The next step is to make the indexes report different numbers so that we can return them as the result of the perfect hash of the elements we are looking up. This is where the second consideration comes handy: we can exploit the second degree of freedom given by the zeros inside the indexes so that all the elements return a different number. Indeed, by flipping a subset of the zeros within the indexes, each index can provide up to 2^z (where $z = b - \text{popcnt}(\text{index}) = \text{no. of zeros}$) different numbers[2]. This problem is an instance of the bipartite graph matching problem (see fig.3.6) which can be easily solved because of the 2^z choices per element that help satisfy the Hall marriage's theorem[76].

We will come back to the theorem after a short discussion on an example describing the idea. In fig.3.6, we present an example of the bipartite matching problem given by the iBF in fig.3.5: the index of the element x is 010, thus we have 2 zeros to flip at will and, in the bipartite graph, we have 4 possible matches (namely $010, 011, 110, 111$); the same goes for y whose possible matches are $001, 011, 101, 111$.

Generally, if p_0 represents the probability of a zero, the mean number of choices per index is:

$$\bar{d} = E[2^z] = \sum_{z=0}^{b} \binom{b}{z} p_0^z (1 - p_0)^{b-z} 2^z = (1 + p_0)^b \tag{3.11}$$

[1]Naturally we could have chosen the right as well

[2]Note that, in a real implementation, we take advantage of the popcnt instruction that computes the number of ones in a register and is available on most architectures.

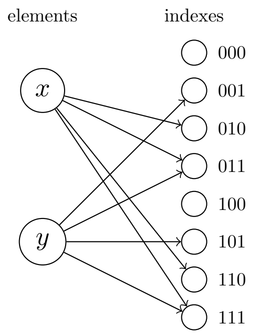


Figure 3.6: The bipartite matching problem.

Of course, for a well-constructed BF, where the number of zeros is the same as the number of ones, the probability p_0 is practically 0.5 and $\overline{d} = 1.5^b$. Since $b \geq \log_2 n$, the mean number of total choices $n \times \overline{d}$ is $O(n^2)$. Therefore the average outdegree of nodes in the bipartite graph is around n. This means we have more links per node than what is needed ($\log n$) to satisfy Hall's theorem with high probability, as shown by Motwani et al. in [77]. Therefore, by means of the Hopcroft-Karp[78] algorithm the bipartite graph matching problem can be easily solved.
In the previous discussion, we have discarded the possibility that two or more indexes could share some bits. For this reason the problem, in real cases, can be highly correlated and NP-hard. Indeed, having always more than b bits between two marked bits is an highly unlikely event, and we are definitely going to have super-positions of indexes: two marked bits closer than b bits imply that their corresponding indexes share at list one bit. This means that if we flip those "colliding[1]" bits, then we are actually affecting the match of two or more elements in the bipartite graph, which leads to large difficulties in the construction.

3.2.2.3 Check and restart

It may happen that the Hopcroft-Karp algorithm may not find any bipartite perfect match. This is mainly due to the choices made in the first step. Because of the complexity of the problem, a totally random choice of the singleton bits in step 1 is not a good idea. In our tests, we experimented that *genetic algorithms* are quite useful in this problem. Because of lack of space we do not include all the details of the genetic algorithm we adopted and do not describe the basics of genetic algorithms (interested readers may look at [79]). However, the main step when adopting genetic algorithm for such kind of problems is the definition of fitness. In our scheme, we associated to each iBF a "DNA" of genes defined by a vector of bits of size m. Such a vector D is such that

$$D \oplus BF = iBF \tag{3.12}$$

(where $\oplus$ is the symbol of a XOR operation) and is adopted by the genetic algorithm as starting point for creating a solution. Basically, we start with a vector D which

[1]Please notice the different meaning of "collision" here that refers to bits shared by more than 1 indexes

is empty (all zeros). Then we create an "individual[1]" by choosing random singleton bits and setting them in D. Note that, setting a bit in D implies clearing a bit in the iBF (as stated by (3.12)). The individual then passes through step 2 and we compute its fitness and store it. Its fitness is basically defined by the number of matched elements. We repeat this procedure for a number of random individuals which form an initial "generation". For each generation, we adopt a roulette-wheel scheme [79], select individuals according to their fitness and couple them, creating new individuals by means of "cross-over" and "mutation" which, in turn, form a new generation. This means we have other choices of singleton bits to be checked and construct the iBF. The procedure is repeated until an individual with maximum fitness (i.e. a perfect match) is found.

Although the algorithm we adopt is quite general, it provides good results in relatively short time. All experimental tests (with $n \leq 2000$ elements) produced a perfect match in less than 5 seconds on a recent Pentium 4 machine. However this procedure is to be performed off-line and its timing requirements are not strict.

3.2.3 Considerations on iBF

Here we introduce a few considerations on iBFs both regarding their size and their speed.

A first observation can be made on the values of parameters m and k: as discussed above in 3.2.2, k must grow like $\log n$ in order to have a good BF w.h.p. This means that m grows like $O(n \log n)$, which resembles the occupancy of Bloomier Filters[75] but with a lower multiplicative factor. Comparing structure sizes (per item) we have:

- a BF requires $m/n = k/\ln 2$ bits;
- a Bloomier filters needs $k \log_2 n/\ln 2$ bits;
- an iBF needs $m/n = \log_2 n/\ln 2$ bits.

Therefore an iBF requires k times less memory than the corresponding (i.e. same number of items) Bloomier Filter , at the cost of a double false positive rate. Indeed, an iBF behaves as a BF with k-1 hash functions in terms of false positives.

On the other hand, an iBF requires a logarithmical amount of memory accesses for each lookup, which is not optimal but effective in many situations, especially when BFs are implemented in network devices and only few changes may be acceptable to the running code or to the hardware description.

Finally, from the point of view of the overall balance of ones and zeros in the structure, we can say that the two construction stages (first mark some bits by clearing them and then add some zeros to the indexes) somewhat compensate each other. Especially if ε is small and b is hence close to $\log_2 n$, the n indexes are going to provide all the combinations of b bits, which means that, within them, a one is as probable as a zero. This is quite important in order to preserve the false positive probability of a BF with k-1 hash functions, as we see in the experimental results.

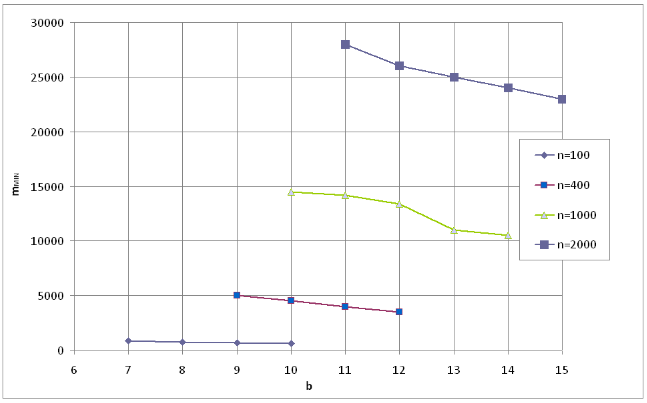


Figure 3.7: Minimal m for the construction of iBF

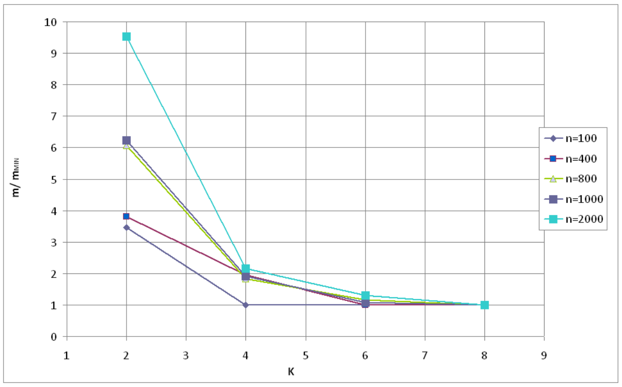


Figure 3.8: Ratio of m over minimal m for the construction of iBF

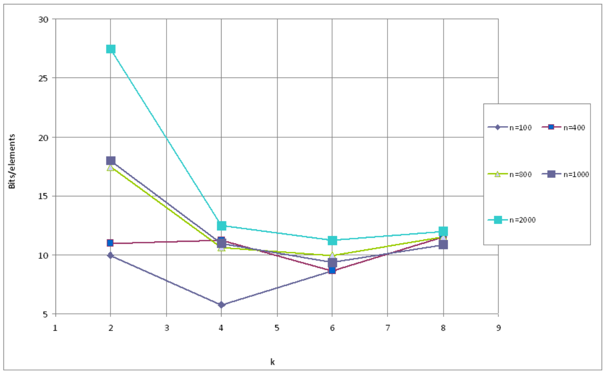


Figure 3.9: Number of bits per element m/n.

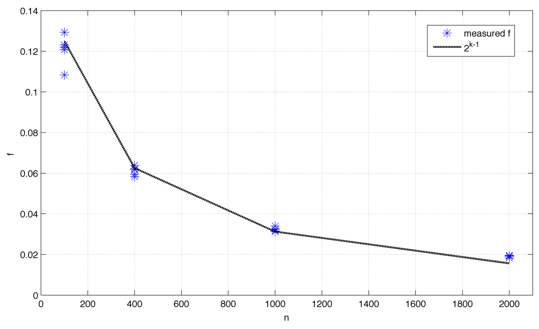


Figure 3.10: False positives in a iBF for $n = 100, 400, 1000, 2000$.

3.2.4 Experimental Evaluation

In the following we show the results of the experimental evaluation of iBFs. We first show in fig.3.7 the effect of the number of output bits b on the size m of the iBF. Enlarging b facilitates the construction of the iBF by allowing smaller structures. Indeed, as b increases, the iBF output grows as 2^b, increasing also the number of links in the bipartite match, which makes the perfect match more probable for small structures. In the graph, values of b are limited as it does not make sense to increase b to value larger than $\log_2 m$. Indeed, if we simple define the output of the iBF as the hash function that points to the singleton bit, we have a fast and simple perfect hash with output domain equal to $m = 2^{\log_2 m}$.
In fig.3.8 we show the behavior of m as k increases. Here the effect of a larger number of choices for a singleton bit is evident as k grows. In the figure, m_{MIN} represents $nk/\ln 2$ which is the value of m that minimizes false positives. As shown in figure, that value of m is also the minimal size of the iBF and it is reachable for values of k that are proportional to $\log n$, as described in sec.3.2.3.
Then figure 3.9 shows the ratio m/n which is the number of bits per element in an iBF. Such a value can be considered as a “cost per unit” for our approach and, again, it reaches its minimal values for values of k which are proportional to $\log n$. This is well justified by the results in the previous figure.
Finally, fig.3.10 shows the amount of false positives we registered by testing the iBF (for $n = 100, 400, 1000, 2000$) with 10 million random queries in 5 different tests. In the graph, the dotted black line represents the theoretical false positive probability (2^{k-1}) and the blue stars are the measured false positive probability value. Measured and theoretical false positive probability overlap for all tested number of elements, confirming our previously stated considerations.

[1]the individual is the genetic term for a possible solution of the problem

Chapter 4

IP-Lookup and Packet Classification

In this chapter, we discuss the results obtained in IP-Lookup and Packet Classification through the use of NetFPGA and Network Processors. IP address lookup is a fundamental task for Internet routers. Because of the rapid growth of both traffic and links capacity, the time budget to process a packet continues to decrease and lookup tables unceasingly grow; therefore, new algorithms are required to improve lookup performance. The process of categorizing packets into flows in an Internet router, on the other hand, is called packet classification. All packets belonging to the same flow obey a predefined rule and are processed in a similar manner by the router itself. For example, all packets with the same destination IP address and protocol may be defined as a flow. Packet classification is the foundation of many Internet functions such as Quality of Service enforcement, monitoring applications, security, and so on. We start introducing three IP-Lookup algorithms based on heuristics to speed up the lookup process in order to be capable of full gigabit linerate. Finally it is shown a novel classification scheme designed for NetFPGA boards which takes advantage of a very compressed version of Deterministic Finite Automata (DFA) in order to process packets at line rate.

4.1 RLA: Routing Lookup Accelerator

Nowadays, on the Internet, the amount of traffic and the speed of links increase day by day. This introduces harder and harder requirements to network routers which have to handle user traffic. In particular, the primary task for routers is IP address lookup, i.e. finding the next hop for packets in the path towards the destination. It requires that a router looks, among its forwarding rules, for the best (i.e., the longest) match with the IP destination address of the packet.

The growth of users and applications, and subsequently of traffic flows, causes an increase of current forwarding tables in size and, in turn, a higher complexity of matching process. On the other hand, the rise in links capacity reduces the time available to process a packet (for instance, on a gigabit link a packet has to be processed within about $0.7\mu s$). For these reasons, IP address lookup is still a bottleneck for networks, thus permanently requiring improvements and new solutions.

Many algorithms have been proposed to obtain high performance in address lookup ([27], [80], [81], [82], [83], [84]). All of them implement Longest Prefix Matching (LPM) schemes, which further exploit several expedients (rules explosion, tracking any rule matched during a search by means of bitmaps) to handle wildcards and partially specified rules.

In this work, we analyze many forwarding tables of real devices: the analysis reveals that the first 16 bits of rules are almost always completely specified and do not present wildcards, i.e. the prefix lengths are greater than 16. Therefore we propose to treat the first 16 bits by means of non-LPM techniques which exploit the complete definition of rules. The target is to accelerate the first phase of lookup, irrespective of the LPM algorithm which is subsequently used.

Moreover, we try to take advantage of the memory hierarchy which characterizes the hardware platforms used for next generation network devices (e.g., network processors). Indeed, such systems provide small on-chip memories with low access latency, therefore reducing memory consumption is a first goal of our approach, in order to use these fast memories only and speed up the overall lookup process. In particular, if we consider only the first 16 bits of IP addresses in the forwarding tables, their values taken as integers present a distribution with large empty spaces, while most of the addresses are grouped around some remarkable peaks. Thus, we propose to divide the address space into different ranges and build a decision data structure for each of them. Then, we encode each address in every range only with the difference with respect to the reference address of that range, thus saving many bits for its representation. Finally, for the lookup in each range, we adopt an adaptive scheme which can provide both direct-addressing and multibit trie search and allows to choose the most efficient data structures.

4.1.1 Related Work

Due to its essential role in Internet routers, IP lookup is a well investigated topic. IP lookup mechanisms encompass trie-based schemes as well as T-CAM (Ternary Content Addressable Memories) solutions and hashing techniques. In the following we report the most important ones, divided in categories, along with a brief description.

4.1.1.1 Content Addressable Memories

Given an input key, a Content Addressable Memory (CAM) compares it against all memory words in parallel; hence, a lookup effectively requires one clock cycle. While binary CAMs perform well for exact match operations, the widespread use of CIDR requires storing and searching entries with arbitrary prefix lengths. Hence, Ternary CAMs were developed with the ability to store an additional “Don’t Care” state thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes at the cost of storage density, access time, and power consumption.

4.1.1.2 Trie-based Schemes

To the best of our knowledge, the most efficient trie-based algorithms for IP lookup are Lulea and Tree Bitmap. For this reason we will take them as references to compare our solution. However, it is important to underline that RLA and the other algorithms can work together: the first 16 bits may be searched with RLA and the remaining ones by using Lulea or Tree Bitmap.
Lulea [80] is based on a data structure that can represent large forwarding tables in a very compact form, which is small enough to fit entirely in the L1/L2 cache of a PC Host or in a small memory of a network processor. It requires the prefix trie to be complete, which means that a node with a single child must be expanded to have two children; the children added in this way are always leaf nodes, and they inherit the next-hop information of the closest ancestor with a specified next-hop, or the undefined next hop if no such ancestor exists. In the Lulea algorithm, the expanded unibit prefix trie denoting the IP forwarding table is split into three levels in a 16-8-8 pattern. The first level consists of a single trie with a depth of 16(notice that it can be completely replaced by our RLA structure) while the second and the third ones consist of a set of smart and compact subtries with a maximum depth of 8. The Lulea algorithm needs only 4-5 bytes per entry for large forwarding tables and allows for performing several millions full IP routing lookups per second with standard general purpose processors. Tree Bitmap [27], which is amenable to both software and hardware implementations, is based on four key ideas:

- all children nodes of a given node are stored contiguously, thus allowing for using just one pointer for all of them;
- there are two bitmaps per node, one for all the internally stored prefixes and one for the external pointers;
- the nodes are kept as small as possible to reduce the required memory access size for a given stride; thus, each node has fixed size and only contains an external pointer bitmap, an internal next hop info bitmap, and a single pointer to the block of children nodes;
- the next hops associated with the internal prefixes kept within each node are stored in a separate array corresponding to such a node.

The advantages of Tree Bitmap over Lulea are the single memory reference per node (Lulea requires two accesses) and the guaranteed fast update time (an update of the

Lulea table may require the entire table to be almost rewritten). Tree Bitmap only requires about 3 bytes per entry and may require a maximum of 4-7 memory references which, if performed in pipeline, allows wire speed forwarding at OC-192 (10 Gbps) rates.

4.1.1.3 Hardware-based Algorithms

Gupta et al. [85] presented lookup schemes (mainly specific for hardware implementations) which require a maximum of two memory accesses for tables of several dozens of MBs that are stored in DRAM (DIR_{24-8}). By adding an intermediate-length table (DIR_{21-3-8}), the structures can be reduced to a dozen of MBs, at the cost of an additional memory access. When implemented in a hardware pipeline, the proposed schemes can achieve one route lookup every memory access. This yields about 20 million lookups per second.
Huang and Zhao [86], instead, introduced a scheme with a segmentation table of 2^{16} entries which is addressed by the first 16 bits of IP addresses. Each entry contains the next hop or, if it "hides" more specified prefixes, a pointer to another structure (called Next Hop Array) that stores the next hops. The overall structure is small enough to easily fit into SRAM memories. The lookup process needs from one to three memory accesses; on a 10-ns SRAM, this mechanism allows approximately 10^8 packets per second.

4.1.1.4 Bloom Filters

BFs are used for LPM in [87]: each BF represents a set of prefixes of a certain length, and the algorithm performs parallel queries on such filters. However, BFs return a yes/no match result (with false positives), thus leaving the lookup job to a priority encoder and a subsequent search in off-chip hash tables.

4.1.2 Motivations

As already mentioned, the aim of RLA is to speed up the lookup of the first 16 bits of the IP destination addresses. An analysis of many forwarding tables of BGP routers, which are available thanks to the Route Views Project of the University of Oregon [88], has highlighted interesting statistical properties, which have led this work. In particular, RLA exploits the following features of the forwarding tables:

- the presence of very few prefixes which have a length less than 16 bits;
- the distribution of the first 16 bits values characterized by empty spaces and spurious peaks.

Tab. 4.1 reports the average prefix lengths obtained through the analysis of the databases from 2001 to 2006. As it is evident, only a very small fraction of prefixes are shorter than 16 bits. This suggests to address the lookup problem for the first 16 bits as an exact lookup instead of a longest prefix matching. Clearly, the lookup rules with prefixes shorter than 16 bits have to be normalized through a process of expansion; however, considering their small number, this does not lead to a significant increase of memory footprint.

Prefix length	Percentage of prefix (%)					
	2001	2002	2003	2004	2005	2006
1-15	0.72	0.71	0.72	0.65	0.7	0.72
16-24	97.86	98.35	97.66	97.43	97.49	97.68
25-32	1.42	0.94	1.62	1.92	1.41	1.6

Table 4.1: Average prefix length distribution for IPv4 BGP table.

Moreover, as shown by the graphs in figure 4.1, which represent forwarding data-bases of different years, the distribution of the first 16 bits presents different areas of concentration over the various ranges of values, thus inspiring our second idea: dividing the address space into several ranges and encoding each address as its difference with respect to the reference address chosen for the range it belongs to. The objectives are: (i) to minimize as much as possible the size of the data-structure and (ii) to speed up the lookup operation by reducing the number of memory accesses. In details, the figure shows the distribution of addresses by dividing the address space in blocks of 8192 values (i.e., the first space represents the value from 0 to 8191, the second one from 8192 to 16383, and so on).

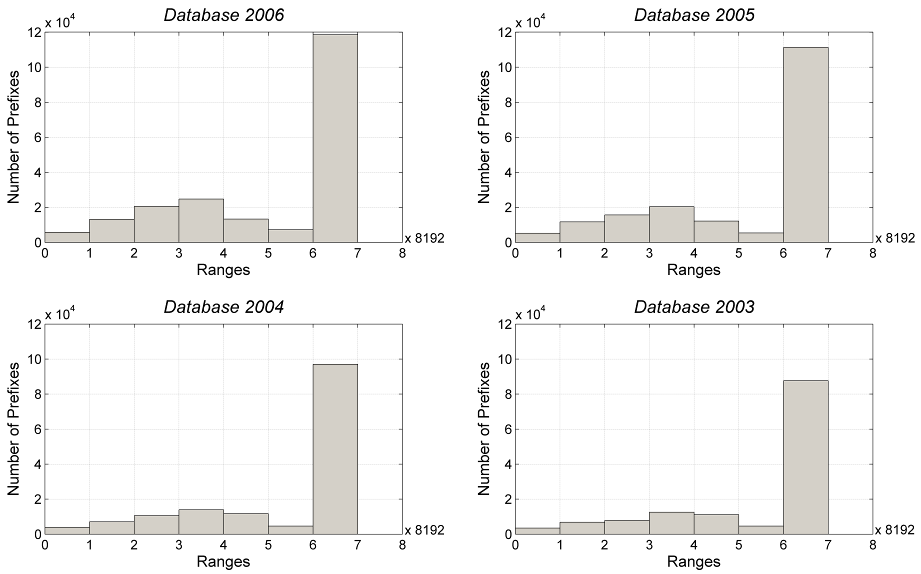


Figure 4.1: Distribution of prefix values belonging to different forwarding tables.

4.1.3 The Algorithm

4.1.3.1 First Step: Range Sub-division

In the first step of our algorithm, the overall address range is split into different smaller ranges. This phase aims only at lowering the memory consumption of the overall structure, while the fine tuning between memory and speed is then committed to the second step (see sec. 4.1.3.2).
In order to perform such a splitting, the first action is to select the maximum number of bits b adopted to encode each address. This parameter affects the range sizes and the number of addresses in each range, thus balancing the tradeoff between the number of structures and their size. Regarding such a tradeoff, it is not obvious that splitting a range in more sub-ranges allows to obtain a memory reduction. Indeed, a single tree on a large range could be more convenient than a set of trees of the different sub-ranges. Therefore, the convenience of a certain value of b cannot be determined a priori, since it strictly depends on the address distribution.
Further, the range subdivision takes into account the empty spaces in the overall address range, in order to use as few ranges as possible and save memory. The pseudocode 8 shows the overall process of range splitting, once the value of b has been fixed. We treat the addresses as integer values and start from the smallest. For each address, we compute the difference with the current reference address: if the number of bits it takes to encode the difference overcomes b, the current address becomes the reference address for a new range. In this way we pursue both the above mentioned objectives: use at most b bits and take advantage of the empty spaces. Notice that the subsequent address encoding could require for a range less than b bits (if the difference between the biggest address and the reference one is less than 2^b).
This process has to be performed for each possible value of b, from 1 to 15 (however the following experimental analysis show that the convenient values are few, thus restricting such a range). After splitting up, for each value of b the overall amount of memory for the all data structures has to be computed and compared to the other ones in order to choose the most convenient value. The selection of the best value for b is an a priori off-line processing, therefore its computational cost is not an issue. However, the experimental runs have shown a non-remarkable amount of processing.

Algorithm 8 Pseudo-code for the range subdivision. N is the total number of addresses X_i and $t[k]$ are the reference addresses.

```
1: $k = 0$
2: $t[k] = X_0$
3: for $i \leftarrow 1, N$ do
4:     $\delta \leftarrow X_i - t[k]$
5:     if $\delta > 2^b - 1$ then
6:         $k \leftarrow k + 1$
7:         $t[k] = X_i$
8:     end if
9: end for
```

All the reference addresses, which univocally identify the ranges, are stored in a small and fast memory. Then, when a packet arrives, it is possible to quickly locate its sub-range G_i through a simple binary search. Then, the packet address is encoded in b_i bits according to the difference with the reference address; such a new value leads

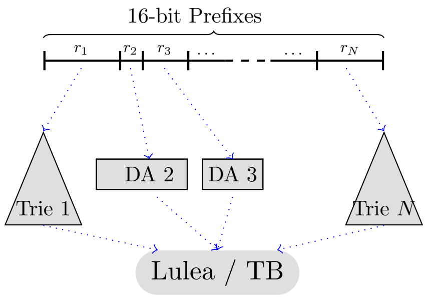


Figure 4.2: Example of lookup by using RLA and then Lulea or Tree Bitmap.

the search in the data structure representing that range. After this first step, the data structure for each range has to be chosen by considering the operational requirements in terms of memory consumption and lookup speed. This is the aim of the second step (sec. 4.1.3.2).

However, in order to show the preliminary advantages of the phase of range splitting and address encoding only, we first perform a gamma of simple experimental runs where we determine the data structures to be used by assuming memory consumption only as the reference metric.

Regarding the data structures, the methods of lookup can be coarsely divided into two techniques: direct addressing (DA) and tree-based. The first requires a single memory access, thus speeding up the lookup operation, but consumes a large amount of memory, while tree-based algorithms obtain memory saving at the cost of more accesses. In particular, in our work we use the multibit trie algorithm as tree-based technique, which performs the lookup by dividing the address in several stride of different lengths, according to an optimization algorithm [89]. By following such an optimization, for ranges with addresses longer than 8 bits, we consider a first stride of 8 bits and a second stride with the remaining ones.

Therefore, in our scheme, multibit trie or DA have to be chosen for the lookup in each range, depending on the specific user requirements (see fig.4.2 where ranges are labeled as $r_1, r_2, \ldots, r_N$). As above mentioned, now we take the memory consumption as the only metric, thus the solution which consumes less memory in the range is simply chosen. Notice that DA may require less memory than multibit tree for particular address distributions.

The graph in figure 4.3 shows the memory required as a function of b in the first step of RLA. The reference forwarding tables are those of the Route Views Project. As evident from the graph, the most convenient choice for b varies between 11 and 13 bits for all the adopted data-sets.

Table 4.2, instead, reports the results of our solution (specifically, the first stage of RLA) in terms of both table construction and lookup process. The forwarding tables are again those of the BGP routers belonging to the Route Views Project and the optimum value of b has been selected from the results of figure 4.3; the number of ranges and memory consumption of table construction have been listed.

The input traffic (i.e., 1 million of 16 bit-long integers per run) has been created by generating 80% of traffic matching the forwarding rules and the remaining 20%

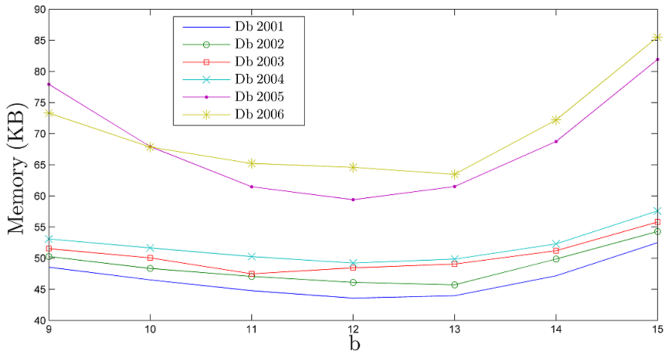


Figure 4.3: First step: the choice of the parameter b.

randomly (which represents a possible distribution of the traffic arriving at a router [90]).

The mean number of accesses have been reported, along with the memory compression with respect to a single multibit trie (which, in the worst case, requires two accesses) and a direct addressing scheme (which requires one access only). The data show that the first step of our algorithm allows for a consistent compression of the lookup data structure. Moreover, the results highlight a further slight improvement in the case of more recent lookup tables and this is a major observation: since our heuristic takes advantage of the properties of the prefixes distribution, these results suggest that our approach is likely to be valuable even in the future. However, the reduction in terms of mean number of memory accesses is quite small; since such a reduction can be achieved when the direct addressing scheme is adaptively chosen for very "crowded" ranges, this suggests that a better strategy for choosing the lookup scheme for each range has to be devised. This motivates the second step.

Year	Entries	Ranges	Memory (KB)	Memory Accesses	Compression (%)	
					Trie	DA
2001	107597	11	43.68	1.97	21.01	66.67
2002	116308	11	45.56	1.98	20.55	65.24
2003	133938	12	47.68	1.99	19.72	63.62
2004	148910	12	49.18	1.99	19.95	62.48
2005	181752	15	60.24	1.85	41.75	64.04
2006	203182	15	63.80	2.00	30.38	61.32

Table 4.2: Performance of RLA and memory gain with respect to multibit trie and direct addressing.

4.1.3.2 Second Step: Table Construction

Preliminary results show the first step of our solution to significantly decrease the memory consumption with respect to a classical single tree approach. However, an intuitive tradeoff between lookup speed and memory consumption exists: the more lookup ranges will adopt direct addressing, the smaller the mean lookup time, at the expenses of an increased memory consumption. Therefore, in the second step the algorithm can be further tuned to meet its operational requirements. In the following, we perform an accurate analysis of this tuning. Let us assume that:

- the size of the lookup ranges (i.e., the parameter b) is fixed, as established in the first phase;
- the address space is divided into R ranges;
- each range requires a different number b_i (never greater than b) to encode the addresses;
- within the i-th range, only K_i different upper byte values are present (that is, some addresses share their most significant byte, which is associated to the first stride of the multibit trie).

A trivial rule to make the decision concerning the lookup scheme to be associated to such a range can be expressed as

$$16 \times 2^{b_i} \lesseqgtr 16 \times K_i(2^8 - 2^{b_i - 8}) \tag{4.1}$$

where the left hand term quantifies the memory consumption in case of a direct addressing scheme, while the right hand term expresses the amount of memory required by an equivalent multibit trie data structure. Since, as already stated, the parameter b_i is fixed, the only variable upon which the decision is based is K_i, for which a threshold value can be easily computed. We point out that, since such a decision rule does not take into account the mean lookup time, the algorithm tradeoff is fairly unbalanced. Furthermore, by taking into examination different lookup tables (associated with different routers and different years) and by computing the value of K_i for different real lookup ranges, we find out that the rule (4.1) leads, in a vast majority of cases, to choose the multibit trie approach, thus limiting the lookup time reduction involved with the opportunistic adoption of direct addressing. In order to increase the possibility of tuning the algorithm, we introduce in (4.1) a penalization parameter α, whose value is set between 0 and 1, to make the adoption of a direct addressing approach more likely:

$$\alpha\left(16 \times 2^{b_i}\right) \lesseqgtr 16 \times K_i(2^8 - 2^{b_i - 8}) \tag{4.2}$$

By adopting such an approach, our algorithm becomes largely tunable, since both memory consumption and mean lookup time can be expressed as a function of the α parameter and, therefore, they can be chosen according to specific requirements. In particular, by indicating with M the memory occupation and with S the lookup speed (expressed in terms of memory accesses) and by defining the parameter $h_i = [2^{8-b_i} + K_i/2^8]$, the following equations hold:

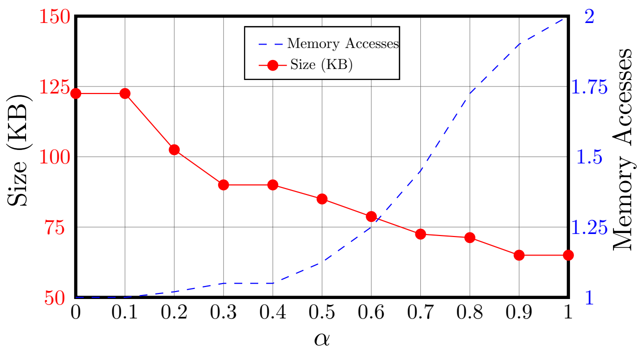


Figure 4.4: Variation of memory consumption and mean number of memory accesses of the RLA algorithm with respect to α for the database of 2006.

$$M = 16 \sum_{i=1}^{R} 2^{b_i} \left[h_i + (1 - h_i) \operatorname{rect}\left(\frac{2\alpha}{h_i}\right) \right] \tag{4.3}$$

$$S = 2 - \sum_{i=1}^{R} p_i \operatorname{rect}\left(\frac{2\alpha}{h_i}\right) \tag{4.4}$$

where $rect(x)$ is the common normalized rectangular function (which is equal to 1 for $-0.5 \le x \le 0.5$ and 0 elsewhere) and p_i is the probability that an address falls in range i. Figure 4.4 shows, for the database of 2006 and for its optimum value $b = 13$, the variation of the two quantities with respect to the penalization parameter α; by examining such a plot, it is possible to tune the algorithm to achieve the desired tradeoff.

It is worth noticing that our algorithm degenerates into several direct addressing structures only, if the value of α falls below the threshold:

$$\alpha < \frac{2^8}{2^b} + \frac{K_{min}}{2^8}$$

where K_{min} represents the minimum number of different upper bytes among the ranges. Instead, a multibit trie approach is used within all lookup ranges in case α exceeds the upper threshold value:

$$\alpha > \frac{2^8}{2^{b_{min}}} + \frac{K_{max}}{2^8}$$

where b_{min} is the smallest number of bits used among the ranges and K_{max} is the biggest K_i.

4.1.3.3 Updates

Performing fast insertions or deletions of forwarding entries is not as important as fast lookup, but it is clearly desirable. The analysis in [91] shows that the mean number of updates per minute in a BGP table is equal to 56. If the only way to handle an update is to rebuild the overall data structure (as in Lulea algorithm), then the router must keep two copies of its routing database at the same time to preserve data consistency. Instead, our algorithm allows for fast insertions or deletions with no need for rebuilding the whole structure. Indeed, since RLA maintains a lookup structure for each range, an update can potentially modify a small database only, thus requiring less time for rebuilding and less memory for data consistency. Unfortunately, a large number of insertions or deletions could change the prefix distribution, and this requires to rebuild more small structures, but simple statistic considerations can confirm that such a condition is very unlikely.

4.1.4 Measurements

	Datab. 2004		Datab. 2005		Datab. 2006	
	Mem.(KB)	Acces.	Mem.(KB)	Acces.	Mem.(KB)	Acces.
Lulea	40	2.99	40	2.98	42	2.98
Tree Bitmap	40	3.98	41	3.95	44	3.89
RLA ($\alpha = 1$)	49	1.99	60	1.85	63	2.00
RLA ($\alpha = 0.9$)	50	1.45	60	1.58	64	1.60
RLA ($\alpha = 0.8$)	50	1.45	61	1.47	65	1.47
RLA ($\alpha = 0.7$)	53	1.34	64	1.29	69	1.26
RLA ($\alpha = 0.5$)	58	1.06	67	1.22	69	1.26

Table 4.3: Performance of RLA compared to Lulea and Tree Bitmap.

This section shows a performance comparison between our complete algorithm and the most efficient solutions for IP lookup: Lulea and Tree Bitmap. The data sets are those of the Route View Project, and consist of hundreds of thousands of rules; the input traffic is generated as explained in section 4.1.3.1. Table 4.3 illustrates the results achieved by the different algorithms in processing the first 16 bits for lookup: the code for Lulea and Tree Bitmap has been derived by analyzing [80] and [27], while for RLA the code runs with different values of α. The results confirm that the parameter α allows to tune the tradeoff between memory occupancy and memory accesses. RLA is able to considerably speed up the lookup of the first 16 bits, by reducing the number of memory accesses up to 65% less than Tree Bitmap and 57% less than Lulea. Although the main purpose of our work is to reduce the number of memory accesses, RLA shows also a memory consumption which is bigger but not far from Lulea and Tree Bitmap. This result is extremely appealing, as nowadays the industry trend leads to the production of fast memories, thus highlighting the importance of fast schemes. Therefore, RLA allows for a higher maximum sustainable packet rate than the other lookup engines. For instance, let us consider that any access to a standard SRAM off-chip memory costs about $7ns$ [34] and that most of the lookup delay is just due to the memory accesses (it is a rough, but effective, first order approximation). In these conditions, for the database of 2006, Lulea and Tree Bitmap are able to process on average the first 16 bits for about 48 Mpps (Mega packets per second) and 36 Mpps respectively, while RLA allows to reach rates from 71 to 113 Mpps depending on the value of α.

4.2 H-Cube: Heuristic and Hybrid Hash-based Approach to Fast Lookup

From extensive studies of real-world lookup tables, it appears that there is a large prefix disparity of density within each lookup table 4.1.2. This suggests that a simple and effective solution to the IP lookup problem can be the adoption of a Heuristic and Hybrid Hash-based technique (we name it *H-cube*), where different data structures store different prefix ranges according to their density. In this way, also the memory hierarchy of high performance hardware platforms, such as network processors, can be exploited. For instance, in our approach we store the most crowded address range in a very compact structure (to be put in a fast small memory). For this purpose we choose a specific Minimal Perfect Hash Function (MPHF) 3.1. Instead, the structures representing the other ranges can be put in slower and bigger memories, thus presenting other requirements (i.e., few accesses).

4.2.1 Motivations and Main Idea

As above mentioned, the main idea of *H-cube* is based on the analysis of the forwarding tables of BGP routers, in order to properly design our overall scheme according to the actual prefix length distributions of the IP destination addresses. We make the common assumption that the traffic, in terms of packets frequency distributions over the rules range, has the same behavior of the rules distribution itself ([87][92]).
In details, the analysis of the forwarding tables available from the Route Views Project of the University of Oregon [88] highlights a very unbalanced distribution, as table 4.1 reports (for the years from 2001 up to 2006). In particular, four big areas of "prefix lengths" can be detected, which present very different density of prefixes: 1-16, 17-22, 23-24, 25-32. The largest part of the rules (on the average, about 63%) falls in the range 23-24. The large disparity of rules distribution among the ranges suggests we can accordingly adopt different structures for them and take advantage of different memories.
As for the reference architecture, we make some general assumptions which are satisfied by many network processing devices (e.g. the Intel IXP Network Processor [35]). In particular, we assume our system to be composed of:

- a standard 32 bit processor provided with a fairly small local memory (say, a few KBs);
- an on-chip fast access SRAM memory block (which we will refer to as scratchpad) with higher storage capacity (in the order of 100 KB) and with an access time of two dozens of clock cycles;

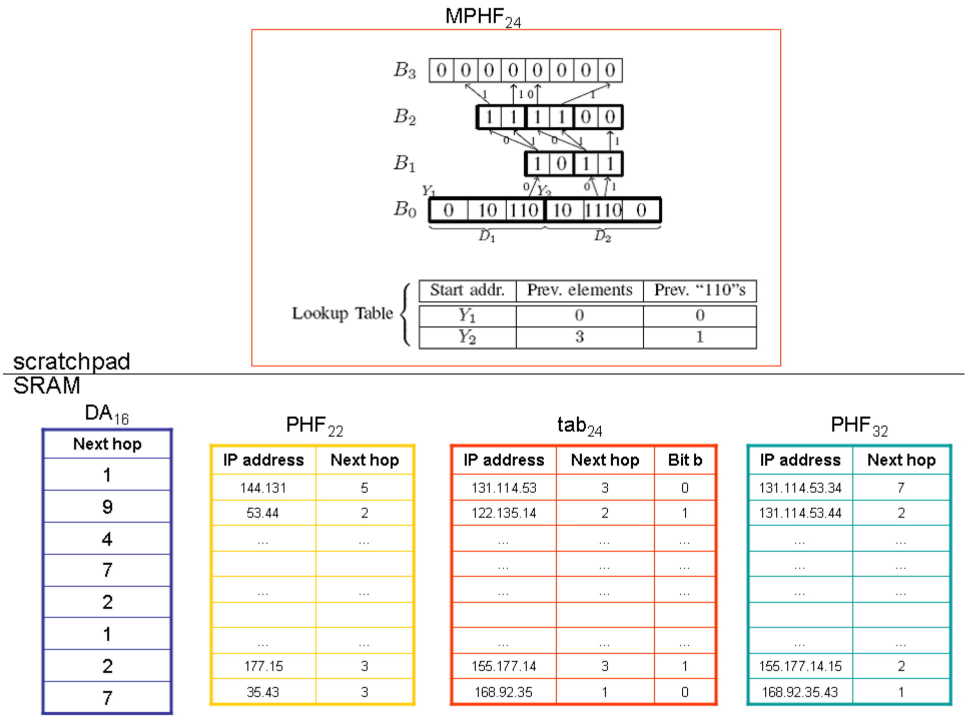
- two off-chip large memory banks: SRAM and DRAM with a storage capacity of dozens of MBs (SRAM) and hundreds of MBs (DRAM) and with an access time in the order of 120 (SRAM) and 250 (DRAM) clock cycles.

From such analysis, it appears convenient to store the table for the 23-24 range in the scratchpad memory, this way reducing the latency for the search in this range, which is by far the most frequent. This requires a very compact data structure and suggests also to start from this table when searching. Instead, the structures representing the other ranges shall be stored in SRAM, therefore it will be better to use searching algorithms which require few memory accesses (while there are not strict size constraints).
While the association of data structures to ranges which we adopt in this work strictly depends on the prefix distributions shown in tab. 4.1, this approach may be generalized as long as the distribution of prefixes presents a large disparity of density.

4.2.1.1 Data Structures

According to the previous considerations, we associate a specific data structure to each range. For the most frequent range, 23-24, we plan to use a MPHF realized by means of Blooming Trees 3.1. This structure (hereafter, we call it $MPHF_{24}$), was accurately described in the previous chapter; it is able to represent many rules in a compact way, thus allowing its storage in the small scratchpad memory. In details, if a match is found for an element, the MPHF gives an unambiguous index which allows to address a table (tab_{24}) where the lookup result (i.e., the outgoing port) is stored. Since the algorithm is "perfect" on the forwarding rules, but the arriving packets can assume any value, a certain probability of collision has to be taken into account; therefore, each entry of such a table stores also the first 24 bits of the relative IP destination address in order to confirm the match.
Let us assume that a k-bit hash function is used and that the original set is composed of N elements: the collision probability can be roughly estimated as $N2^{-k}$. For the 17-22 range, where we have a medium density of rules, we need a structure to be put in SRAM which requires few accesses. Therefore, we choose a Perfect Hash Function (PHF) obtained by double hashing which, again, provides an unambiguous index to address a table (we call it PHF_{22}) with the lookup result. Even in this case, each entry stores the first 22 bits of the relative IP address to solve potential collisions. In the range 1-16, the number of possible rules is limited ($2^{16} \simeq 65000$) and, therefore, a direct addressing table (DA_{16}) can be used, which requires a single memory access. Finally, the least populated range, from 25 to 32 bits, can be implemented through a perfect double hashing or a tree-based scheme (however, we call it PHF_{32}); in any case, we use a data structure requiring a memory access only. Both for the perfect hash functions and direct addressing, an initial rule-explosion is performed. In this way all the prefixes of a range assume the maximum length for that range, thus allowing the representation. Remember that such structures do not have strict constraints in terms of memory; in addition, the tests have shown just a slight memory increase for the explosion.

PHF through Double Hashing The basic idea to create a PHF is using a two-level hashing scheme with universal hashing at each level. In the first level, the n keys



Start addr.	Prev. elements	Prev. "110"s
Y_1	0	0
Y_2	3	1

Next hop
1
9
4
7
2
1
2
7

IP address	Next hop
144.131	5
53.44	2
...	...
...	...
...	...
177.15	3
35.43	3

IP address	Next hop	Bit b
131.114.53	3	0
122.135.14	2	1
...	...	...
...	...	...
...	...	...
...	...	...
155.177.14	3	1
168.92.35	1	0

IP address	Next hop
131.114.53.34	7
131.114.53.44	2
...	...
...	...
...	...
...	...
155.177.14.15	2
168.92.35.43	1

Figure 4.5: A picture of the different structures used in our lookup scheme.

are hashed into m slots by using a hash function h carefully selected from a family of universal hash functions. To handle the collisions in a slot j, a small secondary hash table S_j with an associated hash function h_j is used. By choosing the hash functions h_j carefully, we can guarantee that there are no collisions at the secondary level. However, we will need to let the size m_j of hash table S_j be the square of the number n_j of keys hashing to slot j. While having such a quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirements to be excessive, it has been shown that by properly choosing the first level hash function, the expected total amount of space used is still $O(n)$ [93].

4.2.2 The Algorithm

In this subsection the overall *H-cube* algorithm is accurately explained. By using a sort of binary search, we start by performing the lookup in the most dense range (23-24) and we move towards the other ranges according to this first result.
To be able to use such a binary search [94], in the $MPHF_{24}$ we have to insert some "fake" rules to force, if necessary, the search in the more specified range. Therefore, for each rule in the range 25-32 we insert in the $MPHF_{24}$ a 24 bit-long rule (e.g., the rule 192.168.1.116/30 requires in the $MPHF_{24}$ the rule 192.168.1.0/24). Moreover, each 24 bit-long rule includes a bit b which signals whether more specified rules are present in the PHF_{32}.
By starting from the range 23-24, we could have different cases for a packet (as depicted also in fig.4.6):

- the $MPHF_{24}$ does not give any index for that IP destination address, which means that the packet does not match any rule in the range 23-24 (and also, by construction, in the range 25-32); a further search in the less specified ranges has to be performed;
- the $MPHF_{24}$ gives an index for that IP destination address; we check in the tab_{24} with the following results:
  - it is a false positive: the packet does not match any rule in the ranges 23-24 (and 25-32); a further search in the less specified ranges has to be performed;
  - it is the right entry and the bit b is set: a further search in the PHF_{32} has to be made; if a match is found in the PHF_{32}, this next hop is the correct result, otherwise, the next hop previously found in the $MPHF_{24}$ has to be used;
  - it is the right entry and the bit b is clear: the search is completed (there is no more specified matching rule) and we can use the next hop just found.

If a less specified search is requested, we start by processing the PHF_{22} and then, in case of no match, the DA_{16}.

4.2.3 Theoretical Analysis

In the following we evaluate the properties of *H-cube*.

4.2.3.1 Memory Consumption

Concerning $MPFH_{24}$, its memory consumption S depends on the number n of elements in the set 3.1:

$$S = n \times \left[2 \left(1/2 + \frac{1}{\ln 2} \sum_{i=1}^{L} P_i(\varphi > 1) \right) + 1 + \frac{1}{\ln 2} \right] \tag{4.5}$$

where L is the number of layers required to avoid collisions and $P_i(\varphi > 1)$=1-$P_i(0)$-$P_i(1)$ can be computed from:

$$P_i(\varphi) \simeq \frac{e^{-\alpha_i}\alpha_i^{\varphi}}{\varphi!} = \text{Poisson}(\alpha_i, \varphi) \quad \text{with } \alpha_i = 2^{-i}\ln 2 \tag{4.6}$$

Equation 4.6 claims that the number of elements φ colliding in any block of layer i can be well-approximated by a Poisson pmf with parameter α_i.
The structure tab_{24}, which is addressed by $MPHF_{24}$, requires a number of entries equal to the number of 24 bit-long rules to be represented. Each entry has to store prefix (24 bits) and lookup result (let us assume 4 bits).
Concerning the two lookup structures realized by means of double perfect hashing, we claimed in previous section that the expected total amount of space used is $O(n)$. In particular, [93] shows that, in the worst case, $3n$ slots are needed. Moreover, each slot has to store, besides the lookup result, the overall prefix which represents, in order to handle the collisions which are generated by the elements not belonging to the set.
Finally, for the direct addressing used in the range from 1 to 16 bits, its consumption is obviously 2^{16} locations of 4 bits, for a total amount of 2^{15} bytes. Resuming, we can claim that the total consumption of *H-cube* scheme is $O(n)$.

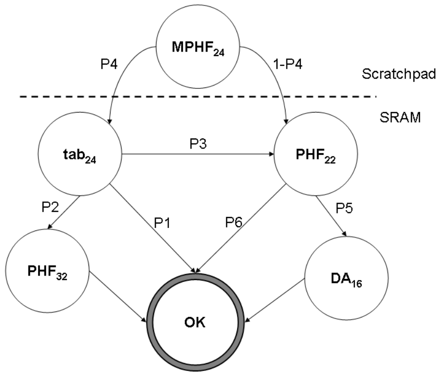


Figure 4.6: Flow diagram of *H-cube* lookup process.

4.2.3.2 Lookup Time

In order to analytically evaluate the performance of our solution, we refer to the flow diagram in figure 4.6. It illustrates the different processing stages a packet may get through during the lookup operation. Each transition in the diagram is labelled with its own probability, that is, the probability of reaching that destination stage conditional that such a source stage has been reached. As a direct consequence of this definition, the probability of a packet passing through a given sequence of processing stages equals the product of the probabilities associated to the corresponding transitions. Notice that the OK block does not stand for an actual processing stage, but simply represents the end of the lookup processing for a given packet.

In addition, during our evaluation, we will consider that a packet does not match any of the rules of the database with probability ϵ; this, in turn, causes the probability of a packet matching a given prefix to be scaled by a factor of $1-\epsilon$.

dataset	num. rules	constr. time (s)	memory consumption (KB)					
			$MPHF_{24}$	tab_{24}	PHF_{32}	PHF_{22}	DA_{16}	tot SRAM
DB1	110000	0.35	34.28	244.88	17.99	239.41	32.76	535
DB3	135000	0.41	41.73	298.11	25.36	323.36	32.76	680
DB5	200000	0.63	63.27	451.93	26.93	518.37	32.76	1030

Table 4.4: Construction time and memory consumption for our scheme.

In the following, we give an analytical evaluation of the transition probabilities on the diagram. P_4 represents the probability of a positive match on the $MPHF_{24}$ structure: this can happen either when the incoming packet actually matches a 23-32 bit-long prefix or in case of a false positive. The former event happens with probability $(1-\epsilon)p(l>22)$, where l is the prefix length of the rule matched by the packet, while the latter happens when a collision in the $MPHF_{24}$ is detected (see section 4.2.1.1) for a packet that does not match the overall rule set or matches a rule shorter than 23 bytes; as a consequence

$$P_4 = (1-\epsilon)p(l>22) + 2^{-k}N(\epsilon + (1-\epsilon)p(l<23))$$

Naturally, a lookup will be performed on the less specified prefixes in PHF_{22} with probability $1-P_4$.

As for P_3, it corresponds to the probability of discovering a false positive after the lookup in the tab_{24}, which can be just expressed as the probability of a false positive scaled by the probability of a tab_{24} lookup (i.e., P_4).

$$P_3 = \frac{2^{-k}N(\epsilon + (1-\epsilon)p(l<23))}{(1-\epsilon)p(l>22) + 2^{-k}N(\epsilon + (1-\epsilon)p(l<23))} \tag{4.7}$$

P_1 is the probability of finding the final next hop value in tab_{24}, which, in turn, corresponds to the probability of the incoming packet matching a 23-24 bit-long prefix. Of course, such a probability has to be scaled by P_4 as well:

$$P_1 = \frac{(1-\epsilon)p(22<l<25)}{(1-\epsilon)p(l>22) + 2^{-k}N(\epsilon + (1-\epsilon)p(l<23))} \tag{4.8}$$

With a similar procedure, the value of P_2, which corresponds to the probability of a lookup on the more specific prefixes in PHF_{32}, can be obtained as:

$$P_2 = \frac{(1-\epsilon)p(l>24)}{(1-\epsilon)p(l>22) + 2^{-k}N(\epsilon + (1-\epsilon)p(l<23))}$$

As for $P6$, it is the probability of finding a match in the PHF_{22} hash table, that is, the probability of the incoming packet matching a 17-22 bit-long prefix scaled by the probability of performing a lookup in the PHF_{22} table:

$$P_6 = \frac{(1-\epsilon)p(16<l<23)}{(1-P_4)+P_3P_4}$$

Of course, P_5, which is the probability of perform a search in the DA_{16}, can be trivially obtained as $1-P_6$.
Based on these results, the mean number of memory accesses required by *H-cube* algorithm can be easily estimated as the sum of number of accesses involved by each possible path from the departure stage (i.e. each viable sequence of processing stages) to the OK block weighted by the probability of such a path. As already stated, the probability of a path is the product of the probabilities associated with each transition of the path, so the mean number of memory accesses $\bar{a}$ could be expressed as:

$$\bar{a} = P_1P_4 + 2(P_2P_4 + P_3P_4P_6 + (1-P_4)P_5) + 3P_3P_4P_5$$

dataset	ε	number of accesses	
		SRAM	scratchpad
DB1	0.1	1.405	2.28
	0.2	1.470	2.28
	0.3	1.535	2.22
	0.4	1.601	2.22
DB3	0.1	1.411	2.4
	0.2	1.481	2.22
	0.3	1.540	2.28
	0.4	1.610	2.1
DB5	0.1	1.412	2.46
	0.2	1.481	2.34
	0.3	1.545	2.34
	0.4	1.609	1.98

Table 4.5: The cost of *H-cube* lookup in terms of memory accesses.

4.2.4 Simulation Results

We simulated the construction of the *H-cube* structure for different databases of rules. We measured both the time required to build the overall structure and the number of memory accesses per packet lookup.
The addresses in the databases are generated by means of the ANSI C function *rand()*,

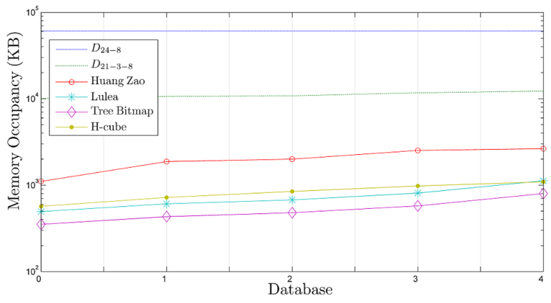


Figure 4.7: Memory consumption for the different lookup schemes.

while prefix lengths follow the actual distributions of BGP routers. The input traffic (i.e., 1 million of packets per run) is also randomly generated; the number of packets with a destination address which does not appear in the database randomly varies from 10% to 40%.

We tested an implementation of our algorithm on an Intel 2.4 Ghz Pentium 4 Core 2 Duo processor. Tab. 4.4 displays, for three random databases, the time and memory required for constructing all the structures, while tab. 4.5 shows the average number of accesses in SRAM and scratchpad for lookup.

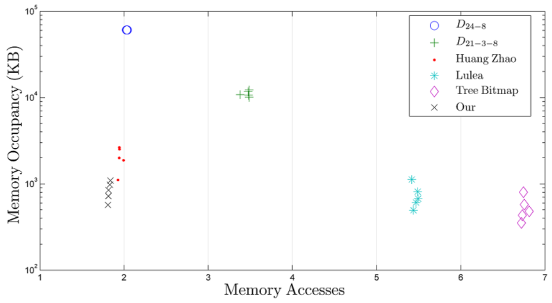


Figure 4.8: Memory occupancy versus number of accesses.

Fig.4.7 and 4.8 compares our algorithm with the most important previous solutions in literature: Lulea, Tree-Bitmap, Huang-Zhao, DIR_{24-8}, and DIR_{21-3-8}. While calculating the mean number of memory accesses, the different speeds of the memory blocks involved have been considered: in particular, each access to the scratchpad memory has been weighted by a factor of 0.17, while each access to DRAM memory by a factor of 2, thus taking into account their different access latency with respect to an off SRAM memory block (as mentioned in 4.2.2). In the graphs, the databases under test are ordered according to the number of rules.

H-cube shows a small memory consumption, as well as Lulea or Tree-Bitmap, while providing a very high lookup speed. Huang-Zhao shows a comparable speed, but at the cost of about twice the memory of our solution.

4.3 A Randomized Scheme for IP Lookup at Wire Speed on NetFPGA

The algorithm proposed in this section is based on data structures called Blooming Trees (hereafter BTs) [56], compact and fast techniques for membership queries. A BT is a Bloom Filter-based structure which allows for memory saving while accepting a certain degree of inaccuracy in data representation. Moreover, it allows to reduce the mean number of memory accesses, which is one of the most important evaluation criteria for the quality of an algorithm for high performance routers, given that it strongly influences the mean time required for a lookup process.
An array of parallel BTs accomplishes the LPM function by storing the entries of the forwarding table belonging to the 16–32 bit range. Every BT has been configured according to the Minimal Perfect Hash Function (MPHF) presented in 3.1, a scheme conceived to obtain memory efficient storage and fast item retrieval. Shorter entries, instead, are stored in a simple Direct Addressing (DA) logical block. In such a module, the address itself (in this case only the 15 most significant bits) is adopted as offset to memory locations.
The reference platform for this algorithm is the NetFPGA [1] board, a new ASIC-based networking hardware which proves to be a perfect tool for research and experimentation. In particular, this work focuses on the data-path of NetFPGA, where the BT-based algorithm for fast IP lookup is implemented. However, the software control plane has been also modified in order to accommodate the management and construction of the novel data structure. These modifications merge perfectly in the preexistent *SCONE* (Software Component of the NetFPGA).

4.4 The algorithm

The algorithms described in sec. 4.1.1 remark the most important metrics to be evaluated in a lookup process: lookup speed, mean number of memory access and update time. Each of the cited solutions tries to maximize general performance, with the aim of be implemented on a high performance router and obtain *line–rate* speed. The main motivations for this work come from the general limitations for high–performance routing hardware: limited memory and speed.
To address these issue, we adopt a probabilistic approach, thus reducing both the memory requirements and the number of external memory accesses. Because of the large heterogeneity of real IP prefixes distribution (as shown in section 4.1.2 and in several works as [87] and [88]), our first idea is to divide the entire rule database into two groups, in order to optimize the structure:

- the prefixes of length ≤ 15, which are the minority of IP prefixes, are simply stored in a Direct Addressing array; this solution is easily implemented in hardware and requires an extremely low portion of the FPGA logic area;
- the prefixes of length ≥ 16 are represented by an array of Blooming Trees (hereafter called BT-array).

In the lookup process, the destination address under processing is hashed and the output is analyzed by the BT-array and the DA module *in parallel* (see fig. 4.9).

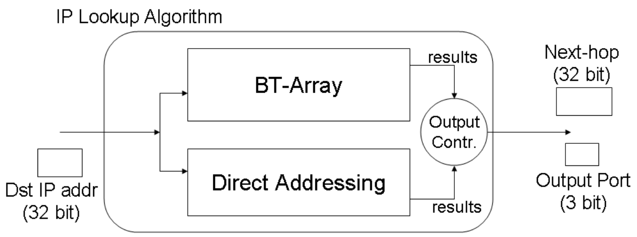


Figure 4.9: The overall IP lookup scheme.

Finally, an *Output Controller* compares the results of both modules and provides the right output (i.e., the longest matching), which is composed of a next-hop address (32 bits) and an output port number (3 bits, given that the NetFPGA has 8 output ports). In the BT-array the prefixes are divided into groups based on their lengths and every group is organized in an MPHF structure (as shown in fig. 4.10). Therefore, the BT-array is an array where 17 parallel queries are conducted at the same time; at the end of the process, a bus of 17 wires carries the results: a wire is set to 1 if there is a match in the corresponding filter. Then a priority encoder collects the results of the BT-array and takes the longest matching prefix, while a SRAM query module checks the correctness of the lookup (since BTs are probabilistic filters in which false positives can happen). In case of false positive, the SRAM query module asks the Priority Encoder for the next longest matching.

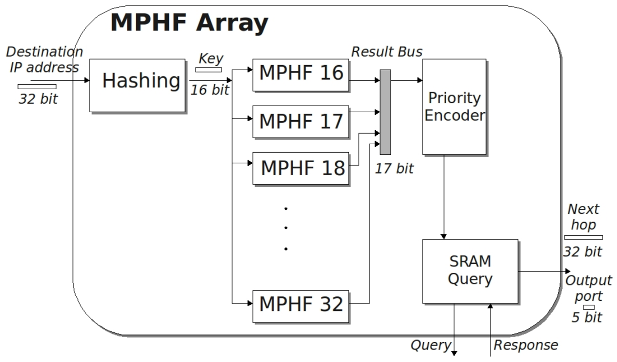


Figure 4.10: BT-array schematic.

4.4.1 Implementation

4.4.1.1 MPHF Module

As above mentioned, the main component of the algorithm is the BT-array, which is composed of a series of MPHFs realized through BTs 3.1. Because of the large difficulties in allocating a variable–sized structure in hardware and for the sake of simplicity, in our implementation we simplify the scheme proposed in 3.1 and adopt a fixed-size structure. In details, the implemented structure presents 3 layers:

- Layer 0: a *Huffman Spectral Bloom Filter* composed of 128 *sections* and with 16 *bins* for every section;
- Layer 1: a simple bitmap that contains *two* bits for every bin of the level 0;
- Layer 2: another bitmap with *two* bits for every bit of the level 1; its size is then of 8192 bits.

These parameters (in terms of number of bins, sections and layers) are chosen in order to allocate, with a very low false positives probability, up to 8192 prefixes per prefix length, which implies that the total maximum number of entries is 128 thousands. Therefore, this implementation can handle even recent prefix rules databases and largely overcome the limitations of the simple (linear-search-based) scheme provided with the standard NetFPGA reference architecture.
In order to further simplify the hardware implementation, each bin of the HSBF consists of 5 bits and its length is fixed. Thus a maximum of 4 elements are allowed at level 0 for the same bin (i.e.: a trailing zero and max 4 bits set to 1). Since the probability of having bins with more than 4 elements is quite small (around 10^{-2}) even if the structure is crowded, this implementation allows for a large number of entries to be stored.
Moreover, a lookup table is used to perform the lookup in the layer 0, which is composed of 128 rows containing the SRAM initial address for each section of the CBF. We place the entire BT and the lookup table in the fast BRAM memory: the HSBF occupies a block of 2048×5 bits, while the lookup table has a BRAM block of 128×20 bits.

4.4.1.2 Managing false positives

As already stated, a BT provides also a certain amount of false positives with probability f. Thus every lookup match has to be confirmed with a final lookup into SRAM. Then, intuitively, the average number of SRAM accesses $\overline{n}$ increases as f grows. More formally, assuming all BTs in the BT-array have the same false positive probability f, we can write:

$$\overline{n} \leq 1 + \sum_{i=1}^{16} f^i \leq \frac{1}{1-f} \tag{4.9}$$

This equation takes into account the probability of the worst case, i.e.: when all BTs provide false positives and are checked in sequence. As one can easily verify, even if f is quite large, the average number of memory accesses is always close to 1 (less than 1.11 for $f = 0.1$).

4.4.2 Results

In this section, the results about the implementation of our algorithm are shown. In details, we focus on the supported bit-rate and resource utilization, in terms of

Table 4.6: Resource utilization for the original lookup algorithm.

Resources	XC2VP50 Utilization	Utilization Percentage
Slices	935 out of 23616	3%
4-input LUTS	1321 out of 47232	2%
Flip Flops	343 out of 47232	0%
Block RAMs	3 out of 232	1%

Table 4.7: Utilization for our algorithm.

Resources	XC2VP50 Utilization	Utilization Percentage
Slices	9803 out of 23616	41%
4-input LUTS	10642 out of 47232	22%
Flip Flops	19606 out of 47232	41%
Block RAMs	68 out of 232	29%

Table 4.8: Utilization for our overall project.

Resources	XC2VP50 Utilization	Utilization Percentage
Slices	17626 out of 23616	74%
4-input LUTS	32252 out of 47232	74%
Flip Flops	31512 out of 47232	66%
Block RAMs	220 out of 232	94%
External IOBs	360 out of 692	52%

slices, 4-input LUTs, flip flops, and Block RAMs. In details, we test our algorithm with different real databases taken from the Route Views Project of the University of Oregon [88] and compare our results with those of the NetFPGA reference router. The NetFPGA reference router does not include the state-of-the-art algorithms in terms of performance and capabilities, as it implements many functionalities primarily as a demo. However, it includes the only available IP lookup engine on NetFPGA which we are aware of. This must be kept in mind while discussing the following comparisons.

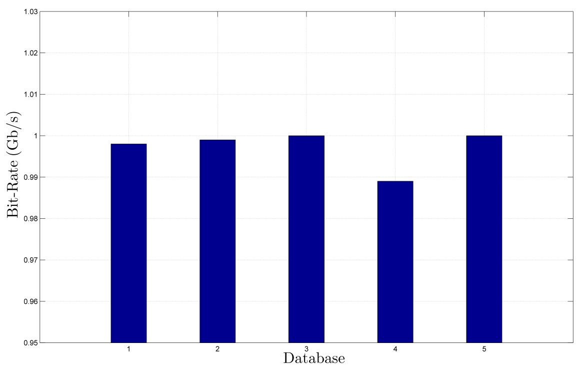


Figure 4.11: Mean bit-rate achieved with different forwarding tables.

Table 4.6 shows the device utilization (both as absolute and relative figures) for the original NetFPGA lookup algorithm. It provides a simple lookup table which allows to manage 32 entries only to be looked for through a linear search. Instead we implement a more efficient and scalable algorithm, which is capable of handling up to around 130000 ($\approx$ 8000 $\times$ 16) entries (by assuming a uniform distribution for entries prefix length). This complexity is obviously paid in terms of resource consumption (see tab. 4.7): in particular, our lookup module uses 41% of the available slices on the Xilinx Virtex II pro 50 FPGA and 29% of the Block RAMs. However, as for the synthesis of the project, it is worth noticing that even though we use a wide number of resources, the timing closure is achieved without any need to re-iterate the project flow. Table 4.8 presents the overall device utilization for the reference router including our lookup algorithm and highlights the extensive use of the various resources. In particular we use 94% of the available Block Rams and 74% of slices and LUTs. In figure 4.11 the behaviour of the NetFPGA router with our lookup algorithm is presented. We take five different forwarding tables of 2009 from the Route Views Project, and we test the functionality of the system by creating synthetic traffic with the Spirent Ax4000 [95], an ASIC-based traffic generator. Due to the false positive effects increasing the number of memory accesses, our algorithm is not able to reach the full line rate; however, the maximum throughput supported is very close to the upper bound (i.e., 1 Gb/s) in all runs.

4.5 On the Use of Compressed DFAs for Packet Classification

The rapid growth of Internet and the fast emergence of new network applications have brought great challenges and complex issues in deploying high-speed and QoS guaranteed IP network. For this reason packet classification has assumed a key role in modern communication networks in order to provide security and QoS. Although packet classification represents one of the most important and critical functions in the process of IP packet forwarding and over the years have been many suggestions in this area of research, none of them can be considered the ultimate solution for all scenarios. Current technology priority, depending on context, reduced memory footprint and high speed rating. This work proposes the use of finite automata to represent the set of rules of the binder. In order to extend the expressiveness of standard classification rules, we chose to address the packet classification problem as a more general pattern matching problem. To this end, we leveraged the existing work on finite state automata (henceforth DFAs). As classification is often performed by either hardware or embedded processors, where memory footprint is an important issue, we have chosen for our scheme the δFA (presented in section 2.1), which presents interesting performance characteristics. In particular, in addition to maintaining a data structure which is much more compact than the standard automation, it needs a lower number of memory accesses than most compressed automata. These properties motivate its use for classification as high-capacity networks, where memory latency is the main factor of performance degradation. In order to implement a prototypal classifier, the hardware platform used is NetFPGA. The aim of this work is to implement a compressed DFA scheme 2.1 (born as PatternMatching engine) on NetFPGA and use it for Packet Classification purpose.

4.5.1 Related Works

Packet classification is an extensively studied topic and several different approaches have been proposed in the literature.

Hardware classifiers traditionally used CAM based techniques. Given an input key, a Content Addressable Memory (CAM) compares it against all of the memory words in parallel; hence, a lookup effectively requires one clock cycle. While binary CAMs perform well for exact match operations, the widespread use of CIDR requires storing and searching entries with arbitrary prefix lengths. Hence, Ternary CAMs were developed with the ability to store an additional Don't Care state thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes at the cost of storage density, access time, and power consumption. A few solutions tried to leverage longest-prefix matching trie-based algorithms (which were conceived for lookup applications) to bi-dimensional matching involving several fields. Such solutions [96] are typically adopted when rules are specified only over destination and source IP addresses. The set-pruning algorithm provides good lookup speed but its memory footprint explodes with the number of rules. A variation of this technique leverages a backtracking primitive [97] to improve memory scalability, at the cost of a significant slow-down, while Grid of Tries [98] fairly balances speed and memory consumption, by speeding up backtracking through the use of switch pointers.

Other solutions leveraged a geometric formalization of the classification problem [99]: as each classification rule can be thought as a range in the multi-dimensional space, classifying a packet means finding out which ranges the corresponding point belongs to. To this end, well-known results from the field of computational geometry can be used.
Another class of algorithms leverage decision trees: although, formally, the algorithm model is analogous to the trie-based approaches, it allows for larger flexibility, as,instead of having all of the relevant fields inspected in a sequential manner, at each node of the tree an arbitrary check can be performed. In particular, Hicuts [81], performs a range check on a particular field while [100] tests single bits. Hypercuts [82] further improbe performance by checking multiple fields at each step. [101] proposed to optimize decision tree by introducing the common branches optimization: rules that, due to wildcards, are assigned to both sons of a decision node, are handled separately, thus reducing worst-case size. [102] Proposes to speed up classification by using a small cache using a set of evolving rules which preserve classification semantics. [103] Partitions the rules into sets which are close to one another in the tuple space, and leverages information from single-field lookups to discard subsets and limit the search space.

4.5.2 Packet Classification as Pattern Matching Problem

The operation of classifying IP packets depending on arbitrary metadata contained in the packets themselves is logically (and practically) equivalent to perform *pattern matching*. Typically, classification rules are expressed in terms of the values of the canonical 5-tuple $SrcIP$, $DestIP$, $SrcPort$, $DestPort$, and $L4 - Protocol$: the output of classification can therefore be obtained by simply applying pattern matching algorithms upon the associated fields of the IP packets. However, our scheme supports classification rules defined over arbitrary metadata (TCP flags are a simple example). As pattern matching is a widely addressed topic in literature, the above observation opens a wide horizon of theoretical and practical solutions to address the problem of packet classification. In recent years, due to the increasing interest focused on *deep packet inspection*, the use of regular expressions (regexes) has become more and more popular because of their high expressiveness in describing sets of strings [16]. Typically, *finite automata* are employed to implement regular expression matching. Deterministic FAs (DFAs), in particular, have gained significant credits as they require one state traversal per character only, although they need an excessive amount of memory as the number of regexes increases. For these reasons, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regular expression sets [17, 18, 19, 20].

4.5.3 Our Solution: Software

The software level of the Classifier takes care of creating and managing the DFA data structures as well as of storing them into the NetFPGA SRAM. The user has to write a simple text file to specify the rules and the associated flowIDs. A bash script then is in charge of calling all the software in order set up the Classifier. First of all it creates the standard (uncompressed) DFA associated to the rules specified by the user. After that it converts the DFA into the δFA structure.

In general, a δFA state consists of a bitmap of 256 bits, which indicate which transition are stored, followed by a list of pointers for such transitions. If the number of transitions in the state is small enough, the bitmap is not an optimal solution, because it would be composed almost entirely of bits to "0". In this case it is more efficient (in terms of memory occupancy) to replace the bitmap with a simple flat list of character-pointer pairs. In figures 4.12 and 4.13 the data structures of the states of type 1 (we refer to the states with bitmap as type 1 states) and type 2 are shown. Each line corresponds to a 72 bits entry. If the state data do not cover the whole row, the entry is padded with 0s and the new state starts at the beginning of the next line. S_1 represents the memory occupancy (in bit) of a type 1 state, while S_2 indicates the memory occupancy of a type 2 state where parameter "n" is the number of specified transitions for a given state. As it is evdident, in a δFA the size of a state is not constant because an arbitrary number of transitions may be stored, depending on the characters whose transitions have to be updated in the local table.

$$S_1 = 360 + 72 * \lceil \frac{n}{3} \rceil \tag{4.10}$$

$$S_2 = 72 + 72 * \lceil \frac{n}{2} \rceil \tag{4.11}$$

If $n \leq 24$, then $S_1 \geq S_2$ and a simple char-transition list is more efficient than a bitmap. For this reason during the creation of the δFA structure, the number of transitions for each state is estimated. If it is than 24 a type 1 state is created, otherwise a character-pointer list (states of type 2) is used.
The state descriptor is a field whose bits have the following meaning:

- Bit 71: if set to 0 it indicates a state of type 1, otherwise type 2;
- Bit 70: if set to 1 it indicates that the state is accepting;
- Bits 69-64: In the type 2 state, they indicate the number of transitions specified. This information is essential to understand where the state ends. 6 bits are sufficient because for this kind of states there are at most 30 transitions. In type 1 states these bits are set to 0.

Type 1 states present also a second byte of information indicating the total number of specified state transitions. This information, wihch is not strictly necessary because it could be derived by counting the total number of bits set in the bitmap, is used to retrieve the data structure through a series of consecutive accesses without having to scan the bitmap. Such bitmap is distributed evenly over 4 rows of 64 bits and an entry contains exactly three pointers.
In type 2 states an entry specifies two transitions, each of them associated with a byte that indicates the corresponding character. The size of type 2 states is from 9 to 144 bytes, while that of type 1 states is from 144 to 819 bytes.

4.5.4 Our Solution: Hardware

The general structure of the classifier is shown in figure 4.14. An optimized version will be discussed in 4.5.4.1. The first operation performed on the incoming packet is

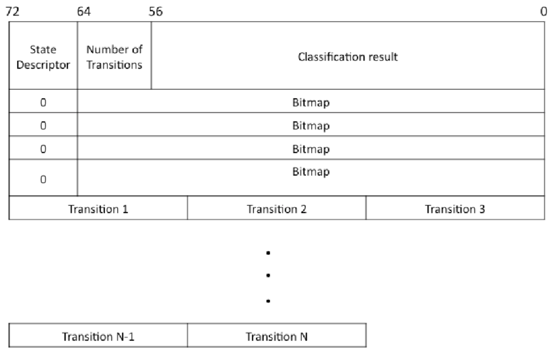


Figure 4.12: Structure of the classifier.

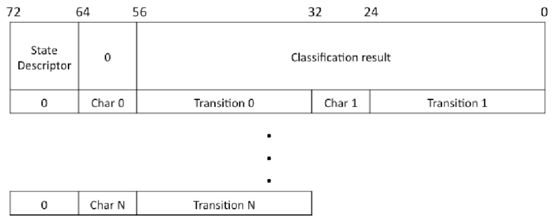


Figure 4.13: Structure of the classifier.

parsing the header fields of the packet in order to compose the string which will be fed into the DFA state machine. The "Datapath Control" block extracts the right fields (i.e.: in the implemented prototype the canonical 5-tuple composed of source and destination IP addresses, layer 4 source and destination port and protocol) and feeds them, one character per time, into the control module of δFA. The "δFA Control Automaton" block contains the FPGA hardware modules that actually implement the automaton logic (i.e.: extract from the SRAM memory the data structure describing the current state, lookup and update the local transition table). This block communicates with the SRAM via a module that masks the access protocol to the memory and requires as inputs only the address of the first entry to be accessed and the number of consecutive entries to be read. "Sram Ctrl" deals with making the appropriate number of SRAM read/write requests to the "SRAM driver".

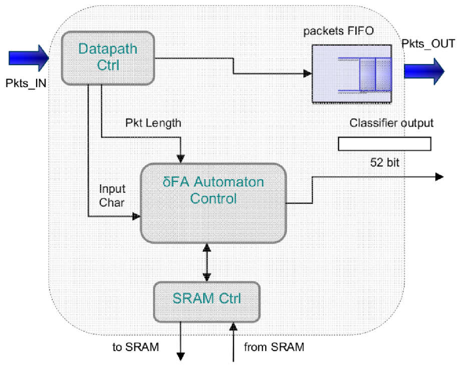


Figure 4.14: Structure of the classifier.

The "δFA Control Automaton" module first performs a single access to SRAM in order to determine which kind of states (i.e.: type 1 or 2) it has to read. The local table maintains the current state transitions that are not stored in SRAM, as they are the same as those of the parent state. In its simplest form is a $256 * 24$ matrix where the i-th row contains the transition (a $24 - bit$ pointer) associated with character i. The table is implemented by using Block RAM (BRAM), a type of memory for quick access, integrated on the FPGA chip. In particular, we use a "dual-port" BRAM in "read-first" mode, which allows to write two entries in the table within a single clock cycle. Notice that, for type 2 states, a dual port configuration is enough to avoid buffering transitions, as at most two pointers are extracted simultaneously form the SRAM. As for type 1 states, since three transitions may be read, one of them would have to be buffered and served in the following clock cycle. For this reason, we chose to deploy two dual port "128x24" BRAMs, each of them containing half the original table. The "BRAM 1" stores the addresses for the even characters while the BRAM 2 those associated with the odd characters. In the worst case all of the pointers may still be stored in the same BRAM block and an intermediate buffer would still be necessary, but in the average case the transitions are divided equally between the tables and therefore can be written in parallel during the same clock cycle, thus speeding up the process of updating the local table.

4.5.4.1 Optimized Classifier

An ordinary way to speed up a classifier is caching flows. Packets of the same flow are likely to exhibit good temporal locality, and the classification result issued for the opening packet can be cached and used for the following ones. Therefore it is useful to introduce a flow-cache, where a new entry is added when the first packet of a new flow enters the system. In this case, the classifier performs a lookup in the classifier table and stores the result in the flow cache. Otherwise, for each packet belonging to a known flow, the classification result is already in the cached data and the amount of memory accesses is reduced. Since the number of flows can be very high, a hash table is an efficient way to implement such a cache. In our current implementation, such a table is kept in BRAM memory.

4.5.5 Experimental Results

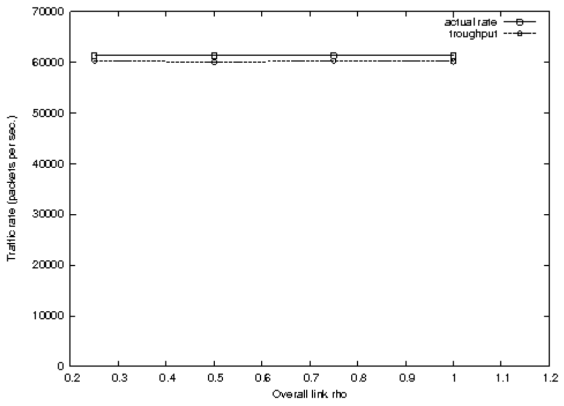


Figure 4.15: Throughtput of the classifier with constant-rate traffic of interest and different rates of the background traffic (ρ stands for link utilization).

In order to assess the performance of our architecture and its capability to filter traffic at line rate, we carried out several tests by using a Spirent AX 4000 hardware based traffic generator; such a device is able to completely saturate a Gigabit link with minimum sized packets, thus recreating the worst case scenario for a netwrok device performing packet-by-packet processing; actually we always performed our tests with minimum sized packets. As the performance of the classifier is strictly dependent not only on the packet rate, but on the number of flows, which, in turn, reflects on the speed-up introduced by the cache, we used the generator API in order to produce a high number of flows. In particular, the AX 4000 generator can inject packets whose addresses are randomly selected within user defined ranges, thus producing traffic where different flows are randomly interleaved. We point out that this scenario is probably more challenging than real traffic, as in the latter packets from the same flows are close to each othe and a cache can provide a significant speed up. In a first experiment, we used the classifier to extract from the traffic a set of 65536 flows matching properly written regexes. The background traffic (which is simply dropped

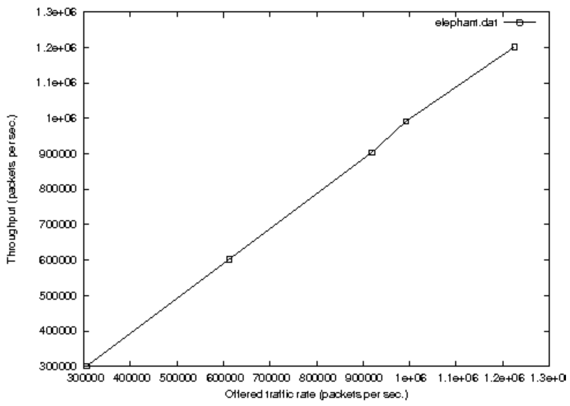


Figure 4.16: Throughtput of the classifier with growing rates of the traffic of interest.

by the classifier after regex matching) is made up of packets whose addresses are chosen within a very large set (thousands of possible source addresses and as many destination addresses) thus potentially providing milions of different flows. We kept the rate of the traffic of interest constant, while gradually increasing that of the background traffic and we measured the rate of the NetFPGA output by using the capturing facilities provided by the Spirent AX. The results are shown in figure 4.15 and apparently the classifier manages to filter in all of the traffic of interest with negligible losses. Besides, the performance is almost constant whatever the rate of the background traffic.In a second experiment we assumed all of the incoming traffic to match the classification ruleset, and we increased its rate until the link was completely saturated. Again, as illustrated in figure 4.16 our classifier is able to process all of the packets with negligible losses.

Chapter 5

Network Monitoring and IP Traffic Generation

In this chapter, we discuss the problems of network monitoring and IP traffic generation. In the last few years, the proposal for measurement-based techniques of traffic engineering and management as well as the continuously increasing concern for network security has raised the interest of researchers and network operators towards the development of measurement tools for traffic monitoring/characterization and to support Intrusion Detection Systems (IDSs). Most of them are designed to run on general purpose architectures and are based on the well known libpcap API, which rapidly became a de facto standard. Even though many improvements have been applied to packet capturing software, it still suffers from several performance flaws, mainly due to the underlying hardware bottlenecks. To overcome these issues, we propose a system architecture based on the cooperation of NetFPGA and a general purpose PC-Host. Finally we introduce BRUNO, a traffic generator we implemented as a tool to test network systems and to push the development of faster applications. This work proposes an hybrid approach, based on a cooperative PC/NP architecture: an advanced software tool runs on a host PC and instructs the processing engines of an Intel IXP2400 Network Processor, which take care of the actual traffic generation. This way we keep the high flexibility of PC tools while achieving the high packet rates of hardware solutions.

5.1 An Open-Source Solution for High-Speed Network Monitoring

Passive network measurement is the best way to observe packets on a network without disturbing the nature or timing of the pre-existing packets. The large availability of flexible, easy to use and easy to customize network monitoring software, suggests the PC as a suitable and cheap platform for network measurement and testing. Indeed, applications such as tcpdump [104], wireshark [105], ntop [106], etc., prove to be very effective and flexible for a large variety of monitoring tasks. As network link speeds increase, it is arguable that traditional network measurement techniques based primarily on software time-stamping and capture of packets will not scale to the required performance levels.
In particular, to sustain a high-packet rate, the PC must drive interface cards by using a polling scheme or interrupt driven I/O with interrupt mitigation enabled. This results into poor timestamp accuracy. In addition, packet loss could happen at high-speed if the host CPU cannot allocate or release memory for packets or if the system bus cannot keep the pace of the incoming data.
Moreover, since the CPU time is used for capturing, no extra CPU power is left for on-line analysis [107] [108]. As result, only off-line processing can often be performed on incoming packets. This is mainly due to the lack of packet processing capabilities on the network interface cards which commonly equip commodity PCs. In particular, the impossibility of:

- time-stamping the arrival of a packet (avoiding interrupt latency);
- filtering unwanted packets out (avoiding memory allocation or release for unwanted packets);
- feeding the host PC with only a fragment of the packet instead of the entire one (avoiding system bus saturation).

make solutions based on reconfigurable hardware more interesting.
The research described in this work addresses the development of a novel measurement tool that overcome the above listed weaknesses of a purely PC–based architecture by using the NetFPGA board [1] in a cooperative platform with a general-purpose PC-Host. The target is to create a powerful and very cheap system that is able to process packets at full rate (Gigabit Ethernet link) with a good timestamp accuracy preserving the flexibility of PC-based solution.

5.1.1 Related Work

Several works on passive measurement systems have been proposed in the literature in the last few years. Our work originates from the need for a flexible and very cheap architecture able to timestamp packets with high accuracy. For this reason we propose a measurement system based on cooperative PC/NetFPGA architecture. A widely recognized benchmarking hardware for traffic capturing is represented by the Endace Dag Card [109]. In this moment, our system supports up to 32 rules, while the DAG 4.3 card, which integrates a simple 7-rule filter, costs as triple as much and the timestamp accurancy it is not so better with respect our system. Moreover, our solution

allows a very flexible and quickly updatable definition of flows. In [110], Wolf et al. propose to use a distributed architecture, called Distributed Online Measurement Environment (DOME), of passive measurement nodes equipped with Intel IXP2400 NP. Their work includes header anonymization schemes and performance is compared to that of Endace DAG 4.3 cards. Both the previous systems are able to analyze up to 500 Mbit/s traffic flows composed by small packets (64 bytes). In [111], Ficara et al. propose an architecture to combine the flexibility of general purpose PCs (equipped with libpcap based applications with the power of Network Processors (NPs) of the Intel IXP2XXX family. In this scenario the NP applies Early Filtering techniques and then it forwards traffic to different sensors, according to Locality Buffers or hash load balancing. This system provides a timestamp accurancy of micoseconds while our solution could compete with the DAG resolution of ten nanoseconds. A related project is "SCAMPI" [112]. This project developed a framework for high speed traffic monitoring and filtering which relies on a FPGA based network adapter. The used board is the COMBO6 [2], developed by CESNET. Such a device implements traffic filtering functions directly on the board and forwards to the upper software layer only the matching packets, thus offloading the kernel from the task of classifying and discarding non-matching packets.
Finally, Luca Deri's nCap [113] and related works provide software-based measurement techniques that work well, but are at the mercy of kernel-based timestamping. These software solutions are both inexpensive and flexible, but traffic load and timestamp quality are both limited by NIC hardware and kernel performance. Hardware solutions typically provide very good timestamp quality, but hardware is typically expensive and offer limited flexibility (especially in the case of proprietary offerings). A NetFPGA-based solution offers the accuracy of hardware timestamping on inexpensive hardware (thanks to support from Xilinx) with the flexibility of open firmware, together with a rapidly growing community of developers and academics.

5.1.2 Architecture

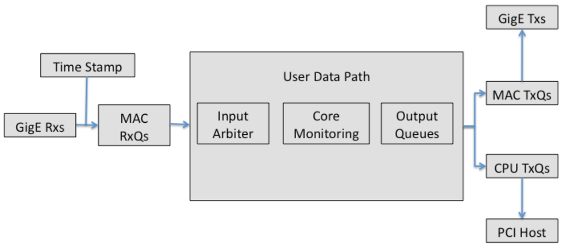


Figure 5.1: The overall monitoring scheme.

A network monitor may either be installed in-series with the link to be monitored, or connected by means of a network tap. Optical network links make the choice is easy: passive optical splitters are inexpensive, and other than during initial installation, offer no possibility of interruption of the link. Copper network links, on the other hand, are more challenging. Some protocols, such as 10/100 Ethernet can be tapped using a passive resistive network but others (including Gigabit Ethernet) require an expensive active tap, such as the NetOptics TP-CU3, or installation of the monitor in-line. In-line monitoring is cheap, and offers the possibility of building an Intrusion Prevention System) system, but comes at the cost of significant extra latency and the risk of interruption of the link, should the monitor lose power, be misconfigured, or otherwise fail. Our aim is to obtain a low cost, high performance and extensible monitoring system. For this reason we decide to install the NetFPGA in series with a link in order to avoid the use of an active copper ethernet tap. Since the NetFPGA has four ports, but supports only copper Ethernet, our monitoring solution integrates the function of an active copper tap by internally coupling two ports of the card. Traffic received on one port is retransmitted out of the other, and visa versa. Where a deployment is especially cost-sensitive, a single NetFPGA-based monitor is sufficient for a single full-duplex link (our solution could be modified to monitor two full-duplex links with ease). Where uptime is more critical, our monitor may also be used with a conventional active copper Gigabit Ethernet tap. As previously stated our solution is based on a cooperation with NetFPGA and the PC-Host in which the board is allocated. While the NetFPGA takes care of filtering the traffic of interest (*i.e.*classifying the flows based on the 5-tuple) and take the timestamp of the packets in a nanosecond accurancy, the developed software (*i.e.*kernel and user space) consists in a set of useful applications that the user could run while the hardware plane is running. The idea behind our system is to take advantage of the fast packet processing capabilities of the FPGA in order to decrease the CPU load. The user, through a simple CLI (Command Line Interface) is able to set what kind of flows he wants to monitor. When the NetFPGA receive a packet, it takes the timestamp and then checks if the packet matches a rule or not. In the first case, it will send a copy of itself, with the associated timestamp, to the CPU through the PCI bus for the host to analyze the data.

5.1.3 Hardware Plane

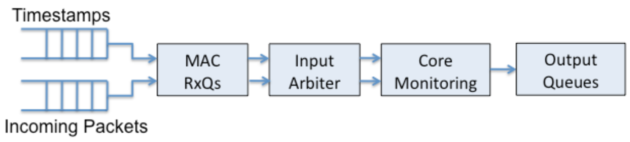


Figure 5.2: The Timestamp and Packet data streaming are passed in parallel.

We organized the system in order that the timestamp data streaming would pass through the modules in parallel with respect to the packet data-streaming (see fig. 5.2). In fact, when a packet arrive at the physical interface, the related timestamp is taken and passed to the next module with an independent data-stream. A strict control system in passing data from one module to the next was added in order to prevent a mismatch between a packet and its associated timestamp. In order to be sure to keep track of every packet that the board receives, we have also inserted two registers that pass to the CPU the timestamp of the packets dropped in the input queues (*i.e.*a packet could be dropped if the associated input queue is full or if a bad

Ethernet CRC is received).

5.1.3.1 Timestamping module: a naive solution

The time-stamping module was added before the MAC fifos, after the RGMII (Reduced Gigabit Media Independent Interface) in order to timestamp the incoming packets as they are received by the hardware. In this way, it is possible to obtain a much lower jitter than if the timestamp were sampled after the kernel receives the packet from the hardware. This module in a first release was implemented as a $64-$bit free-running counter driven by the $125MHz$ system clock, which increments by 8 once every 8 ns. By using the system clock, the time stamp module can be made synchronous with receive logic, and thereby avoid additional error associated with crossing clock domains. When a good frame is received by the physical interface, after the Start of Frame Delimiter (SFD), the "data valid" signal is asserted. We used this signal to mark when a new frame has arrived and to sample the right timestamp of the packet.

5.1.3.2 Obtaining an Accurate Timestamp: DUCK implementation

The timestamp counter implemented in the first release is easy to implement, but provides no means of correcting for oscillator drift and yields data in units of whole nanoseconds. Since standard formats record time in units of seconds, conversion by a floating-point division is required. Both of these drawbacks can be addressed by means of using Direct Digital Synthesis [114], a technique of producing arbitrarily variable frequencies using FPGA-friendly, purely synchronous, digital logic. For these reasons, we implemented the DUCK (Dag Universal Clock Kit) to obtain precise packet timestamp. Refer to [115] for a description of the DUCK operations. The Crystal Frequency used is the one from the GMII receiver path and runs at $125MHz$. Our solution by default produces a time-stamp clock that runs at $2^{26}(67,108,864)Hz$ to provide 26 valid bits of time-stamp fraction. This means that the 64-bit time-stamp increments approximately 15ns every tick. This clock is referred to, in the DUCK implementation, as the Synthetic Frequency (fs), as it is generated from the crystal oscillator on the board. It is also possible to adjust the DDS Rate variable in order to maintain a stable output frequency where the input frequency is unstable. This is very useful as the crystal oscillators used as sources for the Crystal Frequency exhibit jitter and are temperature sensitive. In the current version of the system the core clock of NetFPGA board is used as reference clock for the DUCK (*i.e.*in order to adjust the DDS). This kind of solution in not optimal since the correction will be plagued by the inevitable drift of the clock used as reference. For this reason, in the next version we will use the PPS (Pulse-Per-Second) generated by an external GPS receiver. A synchronization mechanism is obtained through a script that takes the NTP (Network Time Protocol) timestamp and initializes the hardware timestamp counter. At the same time, it writes the $64-$bit value returned by the NTP call in two different $32-$bit NetFPGA host registers.

5.1.3.3 Core Monitoring

Because the NetFPGA PCI interface lacks the bandwidth to record all traffic, we provide a 5-tuple (IP address pair, protocol and port pair) filter. As described in Section 5.1.2, all packets received are retransmitted. Packets that match one of up to 32 filter rules are also copied verbatim, with their timestamp prepended (as shown in figure 5.4), to the host. The timestamp is converted to Intel (little-endian) byte order in the card to save the host most commonly used in these applications from having to do so.

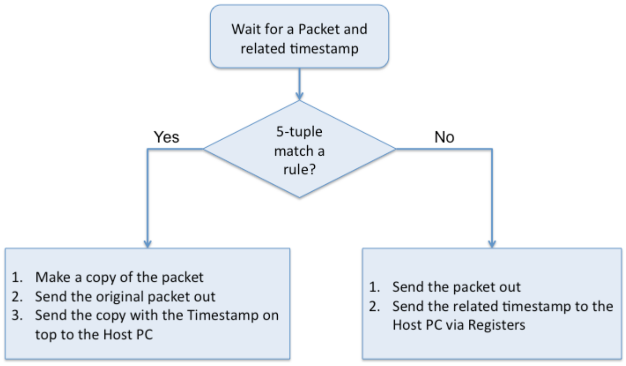

Wait for a Packet and related timestamp
5-tuple match a rule?
Yes
No
1. Make a copy of the packet
2. Send the original packet out
3. Send the copy with the Timestamp on top to the Host PC
1. Send the packet out
2. Send the related timestamp to the Host PC via Registers

Figure 5.3: Flow Diagram of the Core Monitoring.

When a packet arrives, the associated 5-tuple (Source IP, Destination IP, Protocol and 4 layer ports) is extracted and sent to two different TCAM (Ternary Content Addressable Memory) that have sixteen entries each. Since NetFPGA does not have TCAM, we created one using an external Xilinx Coregen application [116]. We chose to implement the TCAMs with SRL16E (16-Bit Shift Register Look-Up-Table (LUT) with Clock Enable) primitives that guarantee a single clock latency on read operations and 16−clock latency on write operations and, as an additional feature, we make sure that we could read and write in the CAM simultaneously, in order that we can change rules on-the-fly without disrupting a capture in progress, or causing packet loss on the network link. As shown in the flow diagram in fig. 5.3, the copy of the arriving packet is made if the 5-tuple matches a rule. The copy is used to pass the packet to the CPU through the PCI interface with the Timestamp (see fig. 5.4) while the original packet is sent out of the board. It is possible to pass to the CPU a maximum of 32 different flows per time. Our implementation doesnt automatically try both combinations of source and destination port and address, requiring two rule slots to specify a complete flow. Rather than try to address this limitation, we feel that a Bloom filter would provide considerably greater density, while also providing the flexibility of specifying only one half of a flow, should that be desirable. Since the object of the filter is to manage PCI interface throughput by limiting irrelevant traffic, any false positive matches from the bloom filters are harmless, and the host can simply throw them away.
If a packet matches no rule, no copy is required. The packet is sent out to the NetF-PGA and the related timestamp is sent to the CPU through two different registers, in

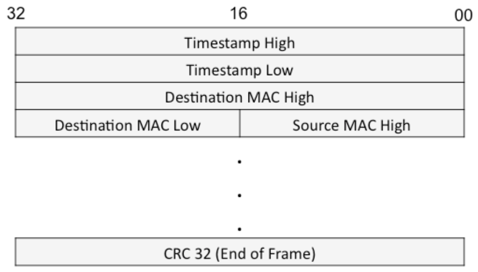


Figure 5.4: Format of the Packet sent to the CPU.

order that the user could keep track of the timestamps of the packets that are filtered by the two TCAMs. Notice that reading the two registers it is possible only to see the timestamp of the last packet that is not copied to the Host.

5.1.4 Software

Libpcap is the de facto standard capture API but, without any change in the NetFPGA kernel module, libpcap applications cannot yet directly be used with our monitoring solution. Simple packet recorders should work, but the 8-byte timestamp prepended to each packet will confound protocol analysis. For this reason we changed the NetFPGA kernel module running on the PC-Host in order to remove the timestamp prepended and store the value in the related structure in the *sk–buff*. To do so, we interfaces the NetFPGA board with a PC-Host running a newer kernel in order to have a time variable in the sk-buff structure that allows nanoseconds granularity (*i.e.*older kernels allows only microsecond granularity).
Such modification allows the use of libpcap for packet capturing using the timestamp taken directly from the hardware NetFPGA. Unfortunately, the limits of the current version of libpcap prevent a resolution of the nanosecond (*i.e.*only microseconds resolution is supported). For this reason the current libpcap also have been hacked in order to support the nanosecond resolution. Finally, in order to guarantee an online traffic analiyisis even the current version of TCPdump has been changed to correctly interface with our libpcap.

Table 5.1: Device utilization for the Passive Monitoring System.

Resources	XC2VP50 Utilization	Utilization Percentage
Slices	19685 out of 23616	83%
4-input LUTS	28017 out of 47232	59%
Flip Flops	22816 out of 47232	48%
Block RAMs	128 out of 232	55%
External IOBs	356 out of 692	51%

We also provide auxiliary command-line tools for TCAM rule management, and the initialisation of the hardware timestamp using the value returned by the NTP system call. Our CLI (Command Line Interface) that can list the rules set in hardware, insert new rules, load a set of rules from a file or clear a specified rule. A rule is presented as the 5-tuple with the associated masks for each field.

5.1.5 Device Utilization

This Passive Monitor System uses 83% of the available slices on the Xilinx Virtex II Pro 50 FPGA. The largest use of the slices are from the Core-Monitoring module where the two TCAMs are instantiated. Almost sixty percent of the block RAMs available are used. The main use of block RAMs occurs in the instantiation of the TCAMs and in the FIFOs used between the modules and the main input and output queues of the system.

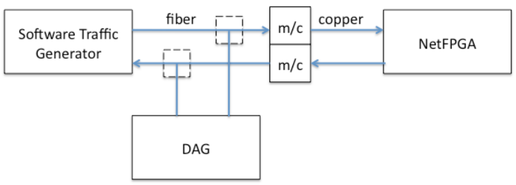


Figure 5.5: System setup.

5.1.6 Results

In the first experiment, we characterised the latency through the NetFPGA with an Endace DAG 4.3ge SX, as shown in figure 5.5. Being optical, we were obliged to use a pair of media converters (Allied-Telesyn AT-MC-1004), and we didnt have the means at our disposal to calibrate out the latency contributed by these devices. We measured latency through the two converters and the NetFPGA card at a constant 2.4 μs, irrespective of whether the test packets matched a filter rule, or how many rules were programmed into the filter. We tested also the quality of timestamps returned to the host against the DAG card, using as timestamping module the “naive solution”. We used TCPreplay with a real traffic trace as “software traffic generator”.
In the set of experimets we compared both the two absolute drift (fig. 5.6) and the relative drift between the two oscillaror (fig. 5.7). We sent 1000 packets and as we can see in fig.5.6 the inter-arrival times of packets recorded by our solution are exactly the same as those achieved with the DAG card. Fig. 5.7 however shows the relative drift of our solution with respect the DAG card. We sent the same 1000 packets of the previous test and as we can see we lose 1.7 milliseconds in approximately 53.7 seconds. Considering that the timestamping module used in this test provides no means of correcting for oscillator drift we obtianed a good accurancy.

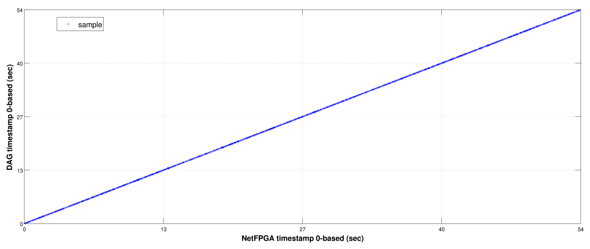


Figure 5.6: Comparison of the two absolute drift with the naive timestamping module.

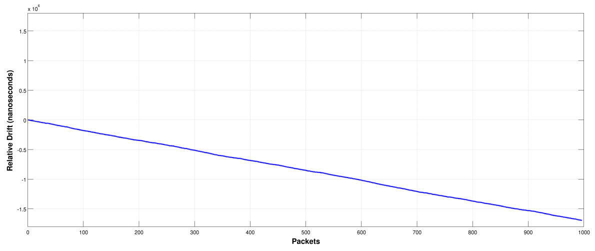


Figure 5.7: Comparison between the two oscillator with the naive timestamping module.

5.2 BRUNO: High Performance Traffic Generator

In the last few years, interest in modern Internet applications has been constantly growing and a significant number of such applications has imposed strict demands on network performance. This has required reliable networks offering high transmission capacity, which in turn has raised the need for network testing to measure performance and reveal possible "weaknesses". Such an evaluation, however, is a very difficult task. Given the high speed of current networks, the simulation of their behavior (for example by means of tools such as the largely diffused *ns2*) [117] is not possible with the proper accuracy: the unavoidable simplifications required by simulations have become unacceptable.

Therefore, the only viable direction to test modern networks is emulation. This requires to generate packet flows which resemble the actual internet traffic, in terms of both data rate and statistical properties. It is obviously a critical task for software tools running on general purpose hardware, especially when dealing with high traffic rates. To address this issue, a very accurate traffic generator, called BRUTE (Browny and RobUst Traffic Engine), has been implemented by our research group [118]. It has a flexible architecture and an extensible design, by providing a number of library modules implementing common traffic profiles. Although BRUTE outperforms all the widespread software tools (KUTE [119], RUDE [120], MGEN [121]) in terms of both the achieved throughput and time precision, it is still limited by PC capabilities in terms of sustainable bit rates (i.e., it is able to generate a maximum traffic load of 400 Mbps).

As already mentioned, the poor performance of software tools is due to the intrinsic limitations of the PC architecture (for which all these tools are designed): such limitations include the PCI bus, which is shared by all the devices thus introducing access conflict penalties, the Operating System, which is usually non real-time, and the adoption of NICs, which are designed and provided with drivers (with rare exceptions) with loose performance standards. Therefore, different solutions have to be devised, and the use of hardware platforms to improve performance and accuracy looks unavoidable. Network processors (NPs) appear as promising solutions for such purposes, and traffic generators based on NP platform are presented in works [122] and [123], which have inspired our activity.
This work presents BRUNO (BRUte on Network prOcessor), a traffic generator built on the IXP2400 Intel Network Processor and based on a modified BRUTE version. BRUTE is designed to run on the PC hosting the NP-card and is in charge of computing departure times according to given traffic models. Then, the host PC writes such information in the memory shared with the packet processing units of NP (i.e., the microengines), which, in turn, use these data to generate packets and send them with the right timeliness. The motivation is a smart distribution of tasks according to capabilities and practicality: while it is very easy to program and make a PC "intelligent" and "flexible" enough to provide new functionalities and models, it is quite difficult to do so on a Network Processor which, in turn, has a great brute-force power to sustain and produce high loads of packet rates. The overall application has shown a sustainable rate of 1 Gbps and a great accuracy in models reproduction, guaranteed by a feedback scheme for time correction, thus confirming the goodness of the design. To date, the traffic models implemented in BRUNO are those inherited from BRUTE (Constant Bit Rate, Poisson, Poisson Arrival of Burst). However, the simple APIs provided by BRUTE are also inherited, so that adding custom traffic models is an easy process that involves a minimum amount of programming skills. In addition, a "playback capability" is also available as BRUNO is able to exactly reproduce a libpcap trace and to introduce a scaling factor on the interarrival times for "speeding up" or "slowing down" the real trace.

5.2.1 Related Works

In the following we briefly introduce some of the most popular and powerful traffic generators developed for PC, FPGA and NP architectures along with related works of interest.
Several open-source tools for traffic generation on commodity PCs have been proposed over the years, most of them designed for the Linux Operating System. KUTE [119] (an evolution of the former UDPgen) is an UDP traffic generator which is designed to achieve high performance over Gigabit-Ethernet. It is based on a Linux kernel module that operates directly on the network device driver bypassing the Linux kernel networking subsystem. This means that its architecture is strictly related to the kernel, and it cannot take advantage of the support of kernel-space extensible interface. RUDE [120], MGEN [121], ITG [124] and BRUTE [118] are user-space tools. The first one is able to instantiate simultaneous patterns of traffic, but it does not provide any explicit support for extensible interfaces and is not suitable to work at high rates, especially with small frames, as shown in [118].

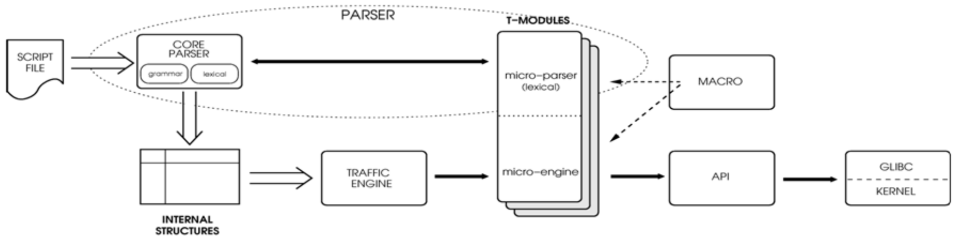


Figure 5.8: Architecture of BRUTE.

MGEN provides both a command line and a GUI for user-friendly traffic generation in user-space. It runs on different Unix-based Operating Systems such as FreeBSD, Linux, NetBSD and Solaris, but its accuracy is limited by the system timers it exploits (e.g.: in the Linux kernel on PC-platforms, the timer resolution used by MGEN is only 10ms [120]).
The Internet Traffic Generator (ITG) [124, 125] aims to reproduce TCP and UDP traffic and replicate appropriate stochastic processes for interdeparture time and packet size. It is based on daemon processes that are contacted through Inter Process Communications by the interfaces. It is able to achieve performance comparable to that of RUDE and MGEN but provides more traffic patterns and runs also under Windows^{TM}.

5.2.1.1 BRUTE

The Browny and RobUst Traffic Engine (BRUTE) [118] takes advantage of the Linux kernel potential in order to accurately generate traffic flows up to very high bit rates. Because of its excellent flexibility due to a simple scripting language and an extensible architecture, it has been chosen as the basis for the development of our generator BRUNO. BRUTE provides extensibility by means of optimized functions and an interface (API) which enable the implementation in C language of additional traffic sources (named *T-modules*) by users. Because of portability issues (at the expense of a slight loss in terms of latency), it uses POSIX.1B FIFO process type and has been designed as an user space application.
Fig. 5.8 shows the architecture of BRUTE:

- the parser reads script files containing the generation requests;
- such information is then stored into an internal database called *mod-line*;
- the traffic engine examines the *mod-line* entries and instantiates the proper traffic handlers, called *micro-engines*, which are defined into the T-modules;
- all the micro-engines are sequentially executed to generate the requested traffic.

Currently BRUTE is available in [126] along with several traffic patterns: Constant Bit Rate, Poisson, Poisson Arrival of Burst, constant inter-departure time, trimodal ethernet distribution and more. The programming script language is organized in a list of statements, each occupying a single line that consists of an optional label, a command identifier and a sequence of parameters of the traffic class. A little example of the script language is reported in the following:

```
lab: cbr msec=1000; rate=1000;
             daddr=10.0.1.10; len=512;
```

This statement instructs the traffic engine to generate a 1 Kfps CBR traffic flow with 512 bytes long frames for a duration of 1000 ms. When not all parameters are specified, BRUTE uses default values (for instance in this example the default source IP address is assumed).
However, all the PC-based generators, even if well designed, are limited by the capacity of the PC architecture. For instance, in a gigabit ethernet scenario, the highest throughput achievable with BRUTE (1.09 Mpps) is reached only in intermittent bursts (as it will be show in section 5.2.5).

5.2.1.2 Hardware architectures for generation

A few solutions for traffic generation upon specialized hardware architectures have been proposed in last years. Abdo et al. [127] employed an Altera Stratix GX FPGA to develop an OC-48 traffic generator. This tool provides high performance but it presents a lack of flexibility and a restricted set of traffic models. This is mainly due to the difficulties in the definition of new models because of the limited programmability. To the best of our knowledge, only two traffic generation tools have been proposed on Network Processors, both for Intel®IXP2XXX NPs. Such tools are reviewed in sec. 5.2.1.2.
This work has been inspired by the need for a generator combining the high flexibility of PC-based tools such as BRUTE and the high performance of dedicated hardware instruments.

Traffic generators on the IXP2400 NP The University of Kentucky developed IXPktgen [122], a generator based on the Intel IXP2400. In spite of the lack of specific informations about this generator, an accurate study of its source code has shown its structure. It employs 4 μEs (working in 8-threads mode) which are used for traffic generation. This implies that, since each thread is statically assigned a single flow, only 32 flows can be generated at the same time. IXPktgen is developed in microcode-assembly and can generate any kind of ethernet frames according to a static parameter file which is read at the startup. Thus the generator is not dynamically reconfigurable.
The Pktgen [123] is a traffic generator proposed by the University of Genova. It is based on the Radysis ENP-2611 board equipped with the Intel IXP2400 NP. It can generate Constant Bit Rate and burst traffic with high throughput. In its design, 5 μEs (working in 4-threads mode with a single flow per thread) are in charge of traffic generations. Therefore it is possible to generate only 20 flows at the same time. The Pktgen code is developed in microC (a C language with several "intrinsics" for IXP-based specific requirements). Although microC compiler is not as optimized as the microcode-assembler (according to Intel's guidelines [10]), the adoption of a C dialect can simplify a possible porting to other platforms. However, as the previous IXPktgen, in this generator the traffic is statically defined and cannot be changed at run-time.

5.2.2 BRUNO

The target of BRUNO is to combine the flexibility of software-based generators with the high performances achievable only by hardware-assisted applications. Therefore in our architecture we exploit both a general purpose PC and an ENP2611 Radisys pci-board equipped with the Intel IXP2400 NP. The DRAM and SRAM memories on the board, accessible through PCI bus, set up the link between PC host and NP in terms of shared data structures.
The user interface, as well as the parsing process and the creation of flow structures, are assigned to the host PC, while the actual traffic creation is committed to the IXP2400. More precisely, the host PC, through an ad-hoc modified version of BRUTE (that we simply call BRUTE in the following), computes departure times and packet lengths according to the user specifications and stores them in the DRAM. On the NP side, a μE named Load Balancer (LB) is in charge of reading data from DRAM and applying a correction algorithm on packet departure times, while 4 μEs named Traffic Generators (TGs) create packets for transmission.

5.2.2.1 Design of BRUNO

In fig. 5.9 the design of our solution is depicted. The first μE, represented by the tagged box on the left (Load Balancer), reads the packet timeline that BRUTE (on the PC) writes in DRAM and SRAM. Then it properly modifies and sends it to the μEs called Traffic Generators, through a ring structure. Rings are circular, fast and small FIFO queues allocated into the scratchpad memory of the IXP2400 [128]. Since the scratchpad memory is the only shared memory that is embedded in the NP, such rings represent an optimal solution for the communication among the processing units. Traffic Generators finally send packet transmission requests to the transmitter (TX) μE, which, in turn, is connected to the Load Balancer through the feedback ring.

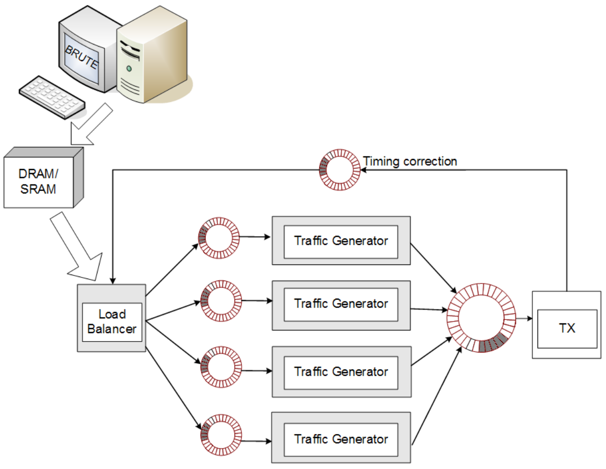


Figure 5.9: Architecture of BRUNO.

The choice of this particular design comes from the need to overcome some limitations that the other NP-based generators have shown. The maximum number of flows that

can be simultaneously generated is one of them. This is mainly due to the fixed association between flows and μE threads. For these reasons, in BRUNO a given flow is not strictly associated to a particular thread, thus allowing for an unlimited number of simultaneous flows.
Moreover, if each thread is in charge of a single flow, it is likely that some threads work more than others, or even that all threads on a certain μE work while other μEs just sleep. This is not desirable since a high number of active threads on the same μE could affect the timeliness of packets and hence the precision of the system. In BRUNO each thread processes packets regardless of flows which they belong to and the LB μE guarantees an equal balance of load among TG μEs, by distributing the packet generation requests in a round robin fashion. This way, for instance, whenever a single flow has to be generated, all the threads in the TGs can work for it.
The feedback ring is introduced in order to improve the traffic generation accuracy. Indeed, this mechanism makes the observed real transmission times available to the LB for a comparison with the ideal departure ones. While the packets request are kept in DRAM in a memory window that is continuously refreshed by the PC with new data, the traffic parameters (e.g. L2 and L3 addresses, as we will see later) are kept in SRAM as they need to be accessed very frequently for the creation of each packet.

5.2.2.2 Load Balancer

The LB μE draws data from DRAM related to a packet generation, properly modifies its departure time and then sends them to a TG μE. Fig. 5.10 depicts the structure for a packet generation request (PR), which is loaded in DRAM by the BRUTE application running on the host PC. The first 32 bits contain the interdeparture time, the packet size (16 bits), the pointer to the flow structure in SRAM (*Flow Index*, 15 bits), and the IP version (IPv4 or IPv6, 1 bit) follow.

Interdeparture Time (31-0)		
Packet Size (31-16)	Flow Index (15-1)	Type 0

Figure 5.10: Structure of a packet request (PR).

More precisely, the threads of LB are divided into two groups.

- Even Threads: they convert the departure times from “relative” (as generated by BRUTE) into “absolute” (as required by TGs) and then move PRs from DRAM to the rings in the local memory of μEs. 8 requests are processed at a time since the DRAM is read in blocks of 16 words of 32 bits.

- Odd Threads: they draw PRs from local memory, adjust the departure times according to the feedbacks (the process of departure times correction is accurately explained in section 5.2.4) and forward the new requests to TGs.

Since the timestamp counter in the TGs is limited to 16 bits, the LB should not send to TGs any PRs scheduled more than 2^{16} clock ticks ahead in the future. Therefore

Output_Port (31-23)	Protocol (22-15)	TOS (14-7)	Ind_Type (6-5)	Res (4-0)
Source_Port (31-16)		Destination_Port (15-0)		
Index (31-24)	Total (23-16)	SRC_ADDs (15-0)		
Index (31-24)	Total (23-16)	DEST_ADDs (15-0)		

Figure 5.11: A flow structure.

odd threads stop when the difference between the time written in the PR and present time is greater than a parameter (*AWAITING_THRESHOLD*), which has to be set at the start of BRUNO application. This parameter can be at most 2^{16} ticks. Since the timestamp counter is increased every 16 clock cycles and the μE clock frequency is set to 600 Mhz:

$$1\,tick = \frac{16}{600 \cdot 10^6} seconds = 26ns$$

The choice of the *AWAITING_THRESHOLD* must be carefully considered. In fact, low values lead to an under-utilization of the Traffic Generators that may not respond properly to abrupt changes in the traffic. On the other hand, high values of this parameter can easily saturate the scratch rings between the Load Balancer and Traffic.

5.2.2.3 Traffic Generators

As shown in fig. 5.9, 4 μEs are designed for packet creation. The thread of Traffic Generators process a packet at a time, by taking the corresponding PR from the communication ring between LB and TG. Through the field *Flow Index* in the PR, the structure describing the flow which the packet belongs to is accessed. Fig. 5.11 shows an example of such flow structures, which are loaded into the SRAM by BRUTE in the initialization phase, according to the user settings.

Output_Port indicates the physical output port for the flow. *Protocol*, *TOS*, *Source_Port*, and *Destination_Port* provide the corresponding fields of L3 and L4 packet headers. *SRC_ADDs* and *DEST_ADDs* point to two SRAM locations which contain a list of source and destination addresses respectively. *Total* provides the number of these addresses, while *Index* indicates the next address to be read if the choice is made in a linear way (otherwise, in random mode, the proper address is suggested by a random number generator). The two bits of *Ind_Type* indicates the selection mode for both source and destination addresses (which in addition can be IP or MAC addresses).

From these data, a thread is able to create packet metadata and L2, L3 and L4 headers. Then the thread is placed in a state of sleep, until the time to send the packet arrives. At this point, the thread wakes up and sends the packet transmission request to the transmission block, which will provide to transmit the packet.

5.2.2.4 Transmitter

The last stage of the generator is the transmitter. The code used is the one provided by Intel®, as optimized for the transmission. Obviously, the feedback system for time correction has been added. It is executed right before before the last step of the transmission process, in order to measure "real departure times" as truthfully as possible.

5.2.2.5 System Initialization

BRUNO requires that the flow structures and the addresses lists are loaded in the SRAM, as well as the requests for transmission generated by BRUTE have to be stored in the DRAM, before the application begins to create traffic. In order to obtain a time consistency among the various μEs, which is fundamental for the good feedback functioning, a system synchronization is required. For this purpose, a specific function is included in Load Balancer, Traffic Generators and Transmitter code to force initialization and timestamp synchronization of the various μEs before the regular functioning.

5.2.3 BRUTE-NP communication

The communication between the BRUTE application, which is in charge of generating the packet requests, and the Network Processor, where the traffic generation actually takes place, is performed through the PCI bus. In particular, both the DRAM and SRAM memory banks on the board are accessible through the local PCI bus, which, in turn, is connected to the PCI bus of the host PC through the Intel 21555 non-transparent PCI-to-PCI bridge [129]. Since the address plan referring to the two buses is different (this is the main reason of using a non-transparent bridging), address translation is implemented on such a device (see fig. 5.12): up to three non overlapping intervals in the PCI address space of the host PC (called downstream windows) can be configured to be translated into the corresponding address intervals in the Radisys PCI address space. Every time the 21555 bridge receives a transaction referring to an address falling into one of the downstream windows, it maps such an address into the corresponding address of the Radisys PCI bus and forwards the transaction over it. In a symmetric way, three upstream windows in the Radisys PCI address space can be defined in order to forward transactions from the board bus to the host PC bus.
Different address translation methods are provided by the bridge, but the most simple and efficient one is the direct base translation: a downstream (or upstream) memory window is defined by a base address, and address translation is performed by simply replacing such a base with a corresponding translated base which defines an address region over the target bus. Since the base length is variable, the size of an address window can be defined by the user: in general, the window size may assume values from 4 KB up to 2 GB, thus allowing to completely map each memory bank of the Radisys board.
The memory translation map can be configured by accessing and setting some control registers associated with the non-transparent bridge; in our implementation, this is done by a Linux kernel module inserted in the host PC operating system. After the initialization of such a module, both the SRAM and the DRAM memory banks

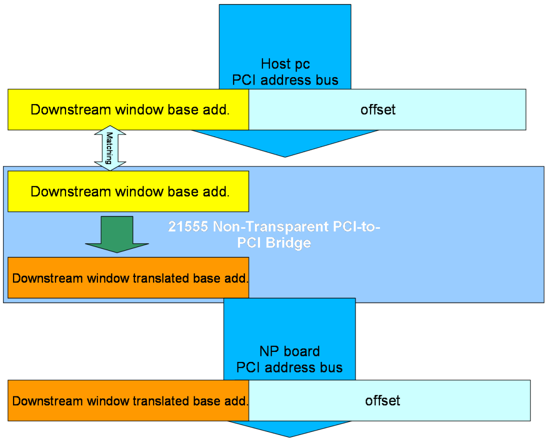

Host pc
PCI address bus
Downstream window base add.
offset
Matching
Downstream window base add.
21555 Non-Transparent PCI-to-PCI Bridge
Downstream window translated base add.
NP board
PCI address bus
Downstream window translated base add.
offset

Figure 5.12: Address Translation.

are accessible as PCI resource regions by the host PC operating system and can be read and written by using system calls referring to memory mapped I/O. In order to offer a simple interface to user applications, our module registers two virtual character devices in the Linux kernel, which are associated to the Radisys DRAM and SRAM banks, respectively. Such devices provide support to the *mmap* access method [130], which allows to register a direct binding between a statically defined physical address region and a user space virtual address region. When a user process accesses a virtual address falling into the mapped area, the virtual memory manager of the kernel directly converts it to the corresponding physical address. This allows user processes to directly access the resources associated with a device, without using any data buffering in the kernel memory. Such a mechanism, which is generally used for accessing high performance devices such as graphical cards, provides the maximum speed for accessing peripheral devices, since no data copying is required. By accessing the two character devices, BRUTE can both configure the parameters defining each generated traffic flow (stored in SRAM) and set the times and lengths for each packet, by writing the corresponding data structure in DRAM.

5.2.3.1 Synchronization

In operational conditions, the DRAM area containing the packet requests must be accessed by both the μEs and the host CPU through the PCI communication mechanism described in the previous section. In order to accurately reproduce the statistical characteristics of a given traffic flow, while the μEs read the metadata and actually generate the packets, new packets lengths and interarrival times must be written in the DRAM memory by the host CPU. Therefore, we need to define a mechanism to cope with the simultaneous presence of readers and writers on the same memory area. In particular, each packet request must be written by the host CPU and read by the NP only once. Let us take as a trivial example a plain Poisson traffic flow: since the packet interdeparture times are independent exponentially distributed random variables, the repetition of a given interdeparture time is not compliant with the flow

specification.
For this reason, a synchronization mechanism between the NP and the host CPU has to be implemented. We choose a method that does not rely on the classic interrupt based solutions (typically used for PCI devices) because of the variable and possibly long latencies which are involved in such schemes. On the contrary, we adopt a polling based mechanism. In particular, in our scheme, the CPU takes care of the whole synchronization task: first, ordinary CPUs are generally faster than our NP (the IXP2400 has a 600Mhz clock rate, while ordinary PCs usually work at a frequency of a few Ghz); in addition, the mean rate at which the data structure in the buffer can be read must be of the same order of the packet sending rate (otherwise, in the long run, the internal buffers of the LB μE, where the packet requests are temporarily stored, would overflow). Since, even at full rate, less than a few million packets per second can be sent by the traffic generator, a common PC can easily fill the buffer faster than it is emptied. Besides, in order to avoid contention issues on the host PC, the BRUTE process can be assigned a higher priority in the Linux processor scheduling mechanism, thus guaranteeing that other tasks interfere only marginally with the traffic generation. As a consequence, there is no need for the NP to wait for the CPU, while it is likely that the CPU has to stop to wait for the NP to read the data in the buffer. In addition, the code running on the NP is optimized for maximum performance and implementing waiting mechanisms could lead to a major performance degradation.

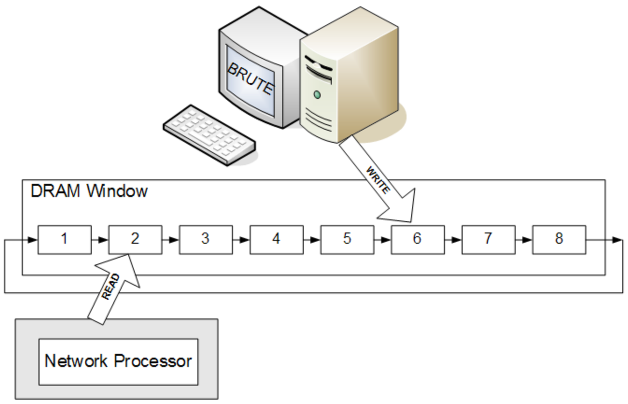


Figure 5.13: DRAM window circular buffer.

The DRAM window containing the packet requests is cyclically read by the Network Processor as a circular buffer. Transferring a large block of data over the PCI bus (and from/to the DRAM) is, in terms of overall delay, more profitable for the host CPU than moving small amounts of packet requests at a time. Therefore, such a circular buffer will be partitioned in blocks containing a given number of data structures (let us say 8); each time either the host CPU or the NP accesses the buffer, a whole block of data structures is read or written. In fig. 5.13, the DRAM window divided into different blocks (arranged in a FIFO circular queue) is depicted.
The NP keeps in its SRAM a pointer to the last block it has read and, in turn, the

CPU maintains in its own memory a pointer to the last block it has written. Before performing a write operation, the CPU reads both pointers to check whether the buffer is full. In such a case, the CPU enters a waiting state for a given waiting time and, after that, it checks the pointers again.
We avoid using a polling mechanism that continuously reads the pointer in order to reduce the number of accesses to the NP SRAM. Indeed, contention for access to the memory block could affect the delay of NP accesses to such a memory, and lead to a performance degradation.
The waiting time must be accurately calculated so as to avoid that the NP reads the whole buffer while the CPU is waiting. Since, as already pointed out, the average buffer reading rate must be equal to the average packet sending rate of the NP, good estimate of such a waiting time can be computed by the host CPU as:

$$T_{delay} = \rho \times \frac{B}{R} \tag{5.1}$$

where B is the overall amount of packet requests that can be contained in the buffer, R is the average packet rate produced by the generator (known by the host PC) and ρ is an arbitrary safety parameter smaller than one (if $\rho = 1$ then T_{delay} is the time needed to empty the whole buffer). For very low values of ρ the CPU floods the SRAM with read requests, thus affecting precision. On the other hand, if $\rho \simeq 1$, the CPU may not be able to fill the buffer properly and the system would not be able to respond in time to abrupt changes of the generated traffic. Preliminary experimental results (limited to the PCI communication) seems to confirm that setting $0.1 \leq \rho \leq 0.4$ is a good choice.
The implementation of the busy waiting mechanism over the host PC relies on the real time capabilities which are included in the original BRUTE. It provides busy waiting functions which, by actually counting the CPU clock cycles and by taking advantage of the Linux process scheduling policies, allow to set a waiting period with a fairly good accuracy.

5.2.4 Performance Evaluation

5.2.4.1 System delays

In this section, we analyze the system behavior in order to estimate the goodness of our design in terms of performance. In particular, we will try to understand if the system is able to generate the maximum packet rate for a gigabit ethernet: 1488000 pps (with 64 bytes per packet). As stated by code simulation, the hardest workload among the μEs is in the charge of the Traffic Generators, so we focus on them.
The mean time (hereafter we call "T" the mean times) spent by a thread of a Traffic Generator μE for its overall processing of a packet is:

$$T_c = T_{r,scr} + T_{el} + T_{w,SRAM} + T_{w,DRAM} + T_{wait} + T_{w,scr} \tag{5.2}$$

where $T_{r,scr}$ represents the mean time spent for reading the PR from the scratchring, T_{el} for processing the request, $T_{w,SRAM}$, $T_{w,DRAM}$, and $T_{w,scr}$ for writing metadata in SRAM, the whole packet in DRAM, and the packet transmission request in the transmission scratchring respectively. Finally, T_{wait} represents the time a thread must

$T_{r,scr}$	T_{el}	$T_{w,SRAM}$	$T_{w,DRAM}$	T_{wait}	$T_{w,scr}$
60	200	100	100	250	60

Table 5.2: Mean times for each operation in clock cycles.

wait when it is placed in the sleep state, as we have described above.
By using Little's law, we compute the available time budget for the overall processing of a packet by the TGs:

$$T_c \cdot \lambda = n \tag{5.3}$$

where λ represents the load (in our worst case 1488000 pps) and n the number of entities that take care of packet generation. In particular, in our design, $n = n_m \cdot n_t$, where $n_t = 8$ is the number of threads per microengine and $n_m = 4$ the number of Traffic Generator μEs. With such values, we obtain a time budget for a packet of $T_c \simeq 12900$ clks.
Then we have measured by means of the Develop Workbench the mean times above listed: tab. 5.2 reports these values in terms of clock cycles. Their sum amounts to 770 clks, which is widely within the computed budget. Therefore our system looks able to support the maximum packet rate in a gigabit ethernet, and the experimental results shown in sec. 5.2.5 confirm this analysis.

5.2.4.2 Timing correction

In fig. 5.14 we represent a schematic view of BRUNO as a system with an input (the ideal departure time $t(n)$ for the n-th packet) and an output (the actual departure time $\tau(n)$). This representation comes in handy in order to describe the actual system implementation and also the timing correction algorithm introduced in the Load Balancer μE. The introduction of a correction algorithm is motivated by the large number of phenomena that, in a complex multi-core system such as our IXP2400 NP, could affect the accuracy of the traffic generation. As an example it suffices to say that the latency of each memory access to any SRAM or DRAM is strictly dependent on the number and the state of all the other threads and the number of requests coming from the PCI bus referring to that memory. A large variety of events (that imply memory and bus accesses) also occurs on the XScale core because of the regular OS house-keeping (e.g.: timing interrupts, memory paging and swapping). All these phenomena may affect a number of packet departures because of their duration in time. Therefore they are modeled in our scheme as a noise $\omega(n)$ with a non-null autocorrelation. In addition, we point out that, since the noise represents a sum of different phenomena that introduce delays, its mean value is positive: $\mathbb{E}[\omega(n)] > 0$. Hence the reason for a correction algorithm ($f(\cdot)$ in fig. 5.14).
However, because of the limited instruction set of the μEs and to limit the delay it introduces, our error-correction algorithm must be devised to be fast and simple, requiring the minimal amount of instructions. Therefore we choose an exponential moving average:

$$\varphi(n) = A \cdot \varphi(n-1) + B \cdot [\hat{\Delta}(n-k) - \Delta(n-k)] \tag{5.4}$$

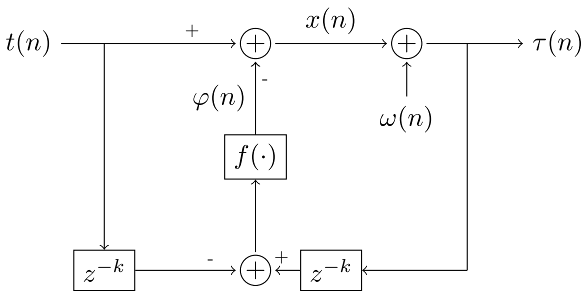


Figure 5.14: Schematic view of BRUNO as a system

where $\varphi(n)$ represents the correction applied to the n-th packet departure time, $\hat{\Delta}(n-k) = \tau(n-k)-\tau(n-k-1)$ is the measured interdeparture time of the $(n-k)$-th packet (taken from the feedback scratchring) and $\Delta(n-k) = t(n-k) - t(n-k-1)$ is the ideal interdeparture time (kept in local memory) of the same packet. The parameters A and B are real and positive numbers, with $A + B = 1$, while the term k takes into account the feedback and system delay (shown in fig. 5.14 as z^{-k}). In fact, when the LB is working on the n-th PR, there are a certain number of PRs in the TGs and on the rings, moreover the feedback mechanism is obviously not instantaneous. Notice that the correction function is applied to the difference of interarrival time rather than on the absolute time themselves: indeed, interarrival times must be very precise while the presence of a possible constant offset between $t(n)$ and $\tau(n)$ is not relevant.

In the following, we assume this term k to be fixed and known and we analyze the system in fig. 5.14 as a discrete-time linear system where packet departure times define the time-domain. Notice that, dealing with a discrete time system that evolves on a packet generation basis (i.e., events are not necessarily equally spaced in time), the mathematical approach is still valid though the frequency parameter cannot be interpreted in the standard way and measured in Hertz.

By simple calculations, it is easy do derive the transfer function $H(z)$ that describes the output of the correcting block as a function of the noise process $\omega(n)$ as:

$$H(z) = \frac{\varphi(z)}{\omega(z)} = \frac{Bz^{-k}(1-z^{-1})}{1-Az^{-1}+Bz^{-k}(1-z^{-1})} \tag{5.5}$$

According to fig. 5.14, and by indicating the impulse response of the system $H(z)$ with $h(n)$, one has:

$$\begin{aligned} \tau(n) &= t(n) - \varphi(n) + \omega(n) \\ &= t(n) + \omega(n) - h(n) \otimes \omega(n) \\ &= t(n) + e(n) \end{aligned} \tag{5.6}$$

The error term $e(n)$ associated with the absolute generated times can be calculated as the output of the system:

$$L(z) = 1 - H(z) = \frac{1-Az^{-1}}{1-z^{-1}+Bz^{-k}(1-z^{-1})} \tag{5.7}$$

which receives as an input the sequence of noise $\omega(n)$.
As above mentioned, though, the error sequence of interest is that of the interarrival time, that is:

$$\begin{aligned}\hat{\Delta}(n) - \Delta(n) &= \tau(n) - \tau(n-1) - (t(n) - t(n-1)) \\ &= e(n) - e(n-1) \\ &= \epsilon(n)\end{aligned} \tag{5.8}$$

In other words, we can express the sequence of errors of interarrival times $\epsilon(n)$ in terms of the noise process $\omega(n)$ through the transfer function of the equivalent system:

$$G(z) = \left(1 - z^{-1}\right) L(z) = \frac{\left(1 - Az^{-1}\right)\left(1 - z^{-1}\right)}{1 - Az^{-1} + Bz^{-k}(1 - z^{-1})} \tag{5.9}$$

The effectiveness of the timing correction mechanism can then be evaluated through the characteristics of the equivalent system $G(z)$.
By assuming the noise process as wide sense stationary, it turns out that the average error is null:

$$\mathbb{E}[\epsilon(n)] = \mathbb{E}[\omega(n)] \cdot G(1) = 0 \tag{5.10}$$

and that its power spectral density $S_\epsilon(f)$ is given by:

$$S_\epsilon(f) = S_\omega(f) \left|G(f)\right|^2 \tag{5.11}$$

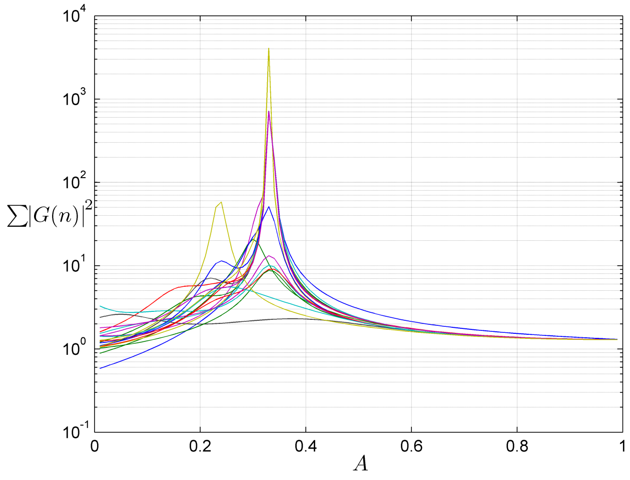


Figure 5.15: Energy of the impulse response of $G(z)$.

In lack of any statistical information on the noise, the selection of parameters A and $B = 1 - A$ should be made in order to minimize the energy of the system $G(z)$ so as to minimize the variance of the error $\epsilon(n)$ in the case of flat spectral density of the noise process.
Fig. 5.15 shows the energy of the system $G(z)$ with respect to A for several values

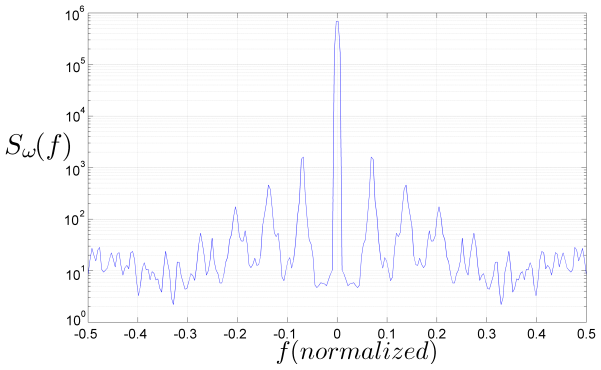


Figure 5.16: Estimated Power Spectral Density of $\omega(n)$.

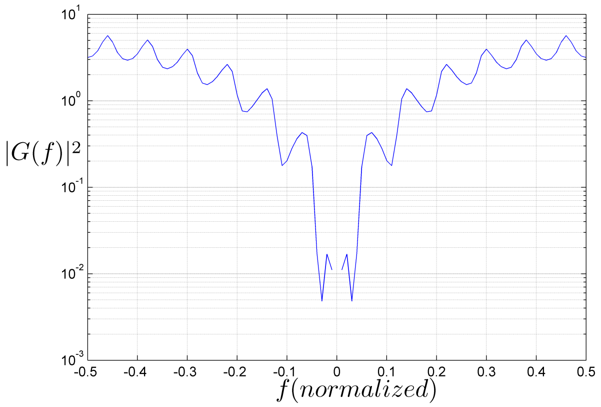


Figure 5.17: Square absolute frequency response of $G(z)$.

of k. From this figure, the choice of $A > 0.5$ seems to be suitable in that the energy approaches 1 and it is very little sensitive with respect to k.
In our experiments (fig. 5.16), though, the noise $\omega(n)$ turns out to be colored as it exhibits stronger components at low frequencies (notice that the peak at $f = 0$ is mainly due to the non zero mean of $\omega(n)$). As shown in fig. 5.17, the frequency response $G(f)$ evaluated for $A = 0.75$ and $k = 12$ (the maximum value of k observed in our experiments) indeed proves a clear high pass behavior, thus effectively filtering out the low frequency relevant components of noise.

5.2.5 Experimental results

We evaluate the actual performance of BRUNO through a wide variety of experimental tests. All the measurements are taken by means of the Spirent AX4000 traffic analyzer [95], which is an ASIC-based tool supporting a precision of the order of nanoseconds. We test the accuracy of BRUNO in traffic models and synthetic traces reproduction, and illustrate the advantages of the time error correction scheme.

5.2.5.1 Traffic models

To date, the library of synthetic models implemented in BRUNO includes three common traffic profiles with different statistical features and parameters. The modular design of the system, however, allows to add any other models upon need. The traffic models implemented are described in the following along with a performance analysis.

Constant Bit Rate traffic. CBR is the easiest traffic pattern that can be generated as it consists of a sequence of packets with constant interdeparture time. In addition to the common parameters used to build IPv4 packets and UDP headers (i.e., IP addresses, packet size, etc.), the only parameter to be specified is given by the *rate*, that is the number of packets sent per second (or, the inverse of the interdeparture time of packets).
In fig. 5.18, which reports the short term packet rate (calculated as a mean over intervals of 0.10 s), we compare the performance of BRUNO to those of BRUTE in reproducing the CBR model. The improvement of BRUNO is evident, in particular for the accuracy and the maximum achievable throughput: BRUNO is able to generate up to 1488000 pps with a high precision. It is worth noticing that this is the maximum packet rate achievable over a 1 Gigabit Ethernet link with the smallest packet size (64 Bytes); this is clearly the worst case scenario for testing network devices. In such conditions BRUTE provides lower throughput and accuracy.

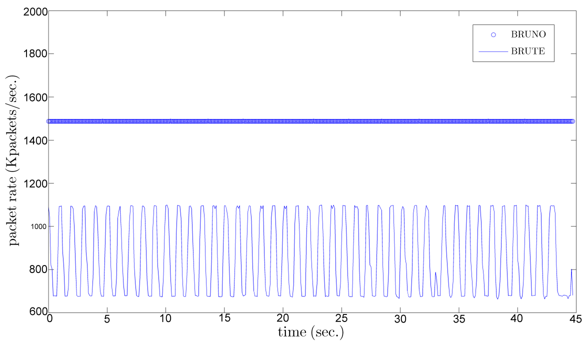


Figure 5.18: CBR traffic: Brute vs Bruno.

Poisson traffic. The Poisson process is historically one of the most popular traffic models and it is obtained by generating packets whose interdeparture times are independent and exponentially distributed random variables. The parameter λ is used to define the mean number of packets generated per second.
Fig.5.19 shows the distribution of a Poisson traffic with throughput= 300 Kpps and $\lambda = 0.03$; the comparison among the traces generated by AX4000, by BRUNO and by BRUTE highlights the improvement of our solution with respect to BRUTE, and especially the capability to properly produce also very small interarrival times (between

0 and 1 μs) thus providing great accuracy at high rates. Moreover, it is worthy noticing that the histogram of interarrival times generated by BRUNO is very similar to that of the commercial AX4000 traffic generator, which is a very expensive hardware solution (hundreds of thousands dollars, while an NP board costs a few thousands dollars).

Poisson Arrival of Burst traffic (PAB). The PAB model is the process given by the superposition of CBR bursts scheduled according to a Poisson process of parameter λ, where the duration of bursts are independent and might be modeled as an arbitrary random variable B with distribution $B(x)$. More formally, the instantaneous rate can be written as

$$R(t) = R \cdot N(t) \tag{5.12}$$

where $N(t)$ is the number of active bursts at time t and R is a scaling factor whose dimension is a data rate (e.g. packet/byte/bit per second). Notice that the random process $N(t)$ is equivalent to the process representing the number of busy servers in a $M/G/\infty$ queue with Poisson arrival rate of parameter λ and service time distribution $B(x)$ (with mean value $\mathbb{E}(B)$).
While the marginal distribution of $N(t)$ is given by:

$$\mathbb{P}\left(N(t) = n\right) = \frac{\lambda\,\mathbb{E}(B)\,t}{n!}\,e^{-\lambda\,\mathbb{E}(B)\,t} \tag{5.13}$$

its correlelation features (and so those of the resulting traffic) vary according to the distribution of burst length $B(x)$. In particular, if burst lengths are distributed according to a power–law distribution (e.g., Pareto distribution), such as:

$$\mathbb{P}\left(B > x\right) = 1 - B(x) = \left(\frac{\theta}{\theta + x}\right)^{\alpha} \tag{5.14}$$

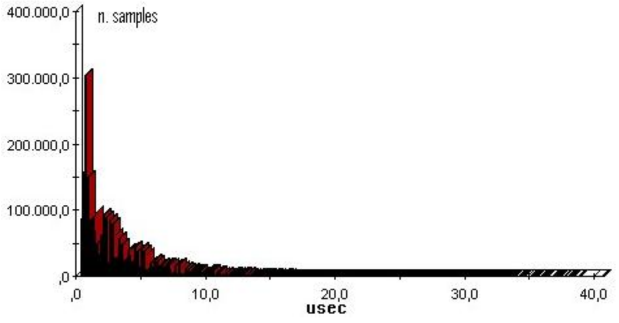
depending on the value of α, the resulting traffic process may exhibit either Short Range Dependence ($\alpha \geq 2$ – light tailed distribution) or Long Range Dependence ($1 < \alpha < 2$ – heavy tailed distribution) [131] with Hurst parameter given by:

$$H = \frac{3 - \alpha}{2} \tag{5.15}$$

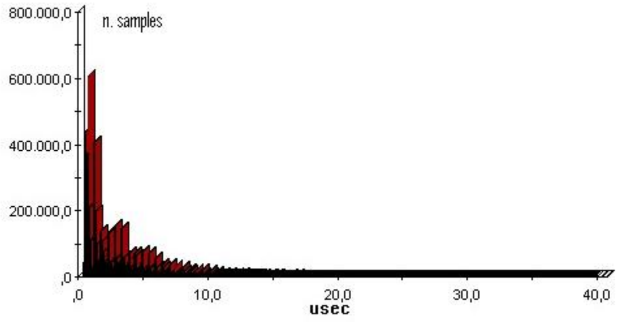
The parameter θ acts simply as used for time offset. Fig. 5.20 and 5.21 show a slice of about 15 minutes of PAB trace generated with parameters $\alpha = 1.5$ (thus $H = 0.75$), $\theta = 1s$, $\lambda = 300$ s^{-1} and $R = 1000$ packets/s. The associated Variance–Time plot clearly proves the presence of Long Range Dependence. Moreover, the estimated value of H is equal to 0.736, which is pretty close to the nominal value $H = 0.75$.

5.2.5.2 Playback capability

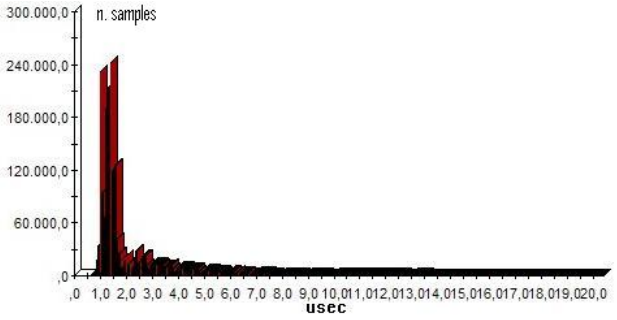
The second set of experimental runs aims at illustrating the "playback capability". Our application is able to exactly reproduce a libpcap trace in terms of packet lengths, IP addresses and ports. Moreover, BRUNO gives the possibility of modifying the original speed of the trace by multiplying its interarrival times by a scale factor: the



(a) AX4000



(b) BRUNO



(c) BRUTE

Figure 5.19: Bar chart of interarrival times of a Poisson traffic ($\lambda = 0.03$).

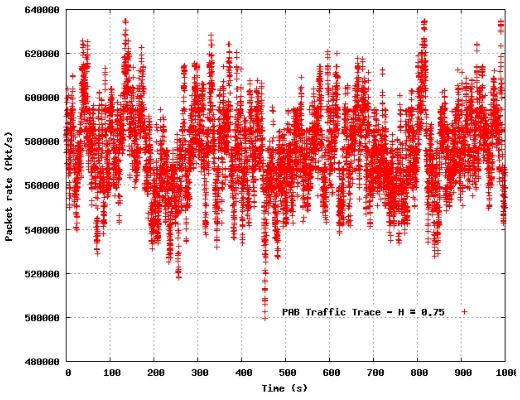


Figure 5.20: PAB traffic profile.

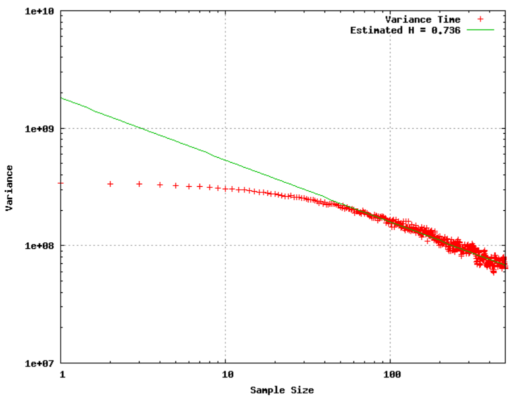
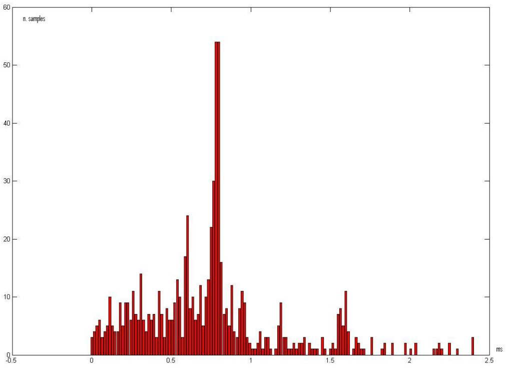


Figure 5.21: Variance time of generated PAB traffic.

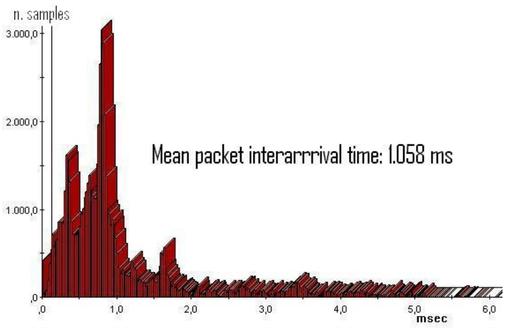
application preserves the time distribution shape of the traffic, while the time scale is “compressed” or “enlarged”. This “playback capability” allows at the same time to perform tests with real traffic and stress devices or networks with different traffic loads.
Fig. 5.22(a) shows the interarrival time distribution of a real SIP call (signalling and data) performed through a soft-phone. Fig. 5.22(b) and 5.22(c) show the distribution of the trace reproduced by BRUNO and an “accelerated” version with a scaling factor of 100. It is evident that the shape is almost the same of the original trace and that the time references (time axis and mean packet interarrival time) differ by the factor of 100.

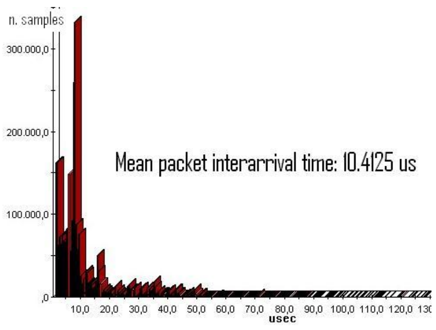
5.2.5.3 Timing Correction Effect

In this set of experiments, the benefits introduced by the error correction mechanism are investigated. We instruct BRUNO to generate CBR traffic flows within a wide range of bit rates, spanning from 100 to 600 Mbps, and we run, for each value of



(a) Original trace (ns)





(b) Trace reproduced by BRUNO (ms)

(c) Trace accelerated by BRUNO (μs)

Figure 5.22: Interarrival times of a 40s SIP call.

rate (Mb/s)	$\sigma_{n\epsilon} - \sigma_{\epsilon}(ns)$
100	14.3
200	12.4
300	3.9
400	3.1
500	14.3
600	13.1

Table 5.3: Interdeparture time variation reduction achieved by the correction mechanism.

bit rate, both the standard version of BRUNO and a modified version in which the feedback correction mechanism was disabled. The measurements are taken again by means of the AX4000.
Tab. 5.3 reports the reduction in the interdeparture time variation due to the introduction of the correction mechanism. We have measured the standard deviations of interdeparture times (σ_{ϵ} for the system with error correction, $\sigma_{n\epsilon}$ for the simple one), obtaining for both versions extremely small values (in the order of hundreds of nanoseconds). However, the use of the timing correction mechanism increases the performance of the system, as shown by the difference $\sigma_{n\epsilon} - \sigma_{\epsilon}$. In any case, these benefits are more evident with increasing complexity of the traffic model generated, because in a CBR model there are few variable factors that can cause a consistent deviation from the ideal behavior, and, therefore, the variance reduction achievable with an error correction mechanism is limited.

Conclusions

This thesis discusses a number of solutions to some taks that represent a critical processing for network devices: IP-Lookup, Packet Classification and Deep Packet Inspection.
We introduced the two reference platforms used both for the tests and implementation of many of the proposed algorithms: NetFPGA and Intel IXP2400 Network Processor. Then we moved into the Deep Packet Inspection section by describing some exact solutions based on a compressed version of DFA (Deterministic Finite Automata) and an approximate one based on Counting Bloom Filters. We discussed two new methods to create Perfect Hash Function with limited amount of memory and we use one of them for some novel IP-Lookup algorithms. We introduced, also, a novel Packet Classification algorithm based on DFAs and we tested it on NetFPGA board. Finally high speed network monitoring with accurate timestamping and IP traffic generation have been discussed with two solution: the former implemented on NetFPGA, the latter on Intel IXP2400 Network Processor.

Appendix A

Hash Functions and Bloom Filters

A.1 Hash Functions

Hashing is one of the most adopted primitives in network devices and regular computers. Its use is quite ubiquituos when dealing with tables which must be addressed by large indexes.
A hash function is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array (cf. associative array). The values returned by a hash function are called hash values, hash codes, hash sums, checksums or simply hashes.
Hash functions are mostly used to speed up table lookup or data comparison taskssuch as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, and so on.
A hash function may map two or more keys to the same hash value. In many applications, it is desirable to minimize the occurrence of such collisions, which means that the hash function must map the keys to the hash values as evenly as possible. Depending on the application, other properties may be required as well. Although the idea was conceived in the 1950s. the design of good hash functions is still a topic of active research.

A.2 Bloom Filters

A Bloom Filter (BF) is a simple data structure for information representation and query processing. It is a randomized method based on hash functions; thus, it allows for false positives, but the space savings often outweigh this drawback. BFs were introduced by Burton Bloom [132] in the 1970s for database applications, but recently they have received a great attention also in the networking area [133], for collaborating in overlay and peer-to-peer networks, packet routing, and measurements. BFs are also proposed for many distributed networking protocols: for example, in order to share

web cache, a proxy periodically broadcasts BFs that represent the contents of their cache. In this situation, BFs are not only data structures but also messages being transmitted in a network.
Thus, several performance parameters have to be taken into account: the probability of false positives, memory size, number of items to be managed and transmission size. BFs do not address the issues of inserting and deleting items in the set. For example, a set may change over time, with elements being inserted and deleted. Deletion cannot be done by simply reversing the insertion operation, because of the collisions created by the hash functions. In order to allow these operations, Counting Bloom Filters have been designed [48]. They are based on the same idea of BFs, but they use fixed size counters (also called bins) instead of single bits of presence. When an item is inserted, the corresponding counters are incremented; deletions can then be safely done by decrementing the counters. CBFs present the problem of counters overflow, which has to be considered in the design.
A Bloom Filter represents a set S of n elements from a universe U by using an array of m bits, denoted by $B[1], ..., B[m]$, initially all set to 0. The filter uses k independent hash functions $h_1, ..., h_k$ with $\log_2(m)$ bits long output, that independently map each element in the universe to a random number uniformly distributed over the range. For each element x in S, the bits $B[h_i(x)]$ are set to 1, for $1 \leq i \leq k$ (a bit can be set to 1 multiple times). To answer a query of the form *"Is y in S?"*, we check whether all $B[h_i(y)]$ are set to 1. If not, y is not a member of S, by construction. If all $B[h_i(y)]$ are set to 1, it is assumed that y is in S, hence a BF may yield a false positive. The probability of a false positive f can be tuned by choosing the proper values for m and k. It is a well-known result [48] that the minimum f is obtained for $k = (m/n)\ln 2$. In this configuration, all bits $B[1], ..., B[m]$ are set or cleared with probability $p = 1/2$ (thus, roughly, the same number of ones and zeros are present in the BF).

References

[1] *http://www.netfpga.org.* 3, 130, 144

[2] *http://www.liberouter.org/hardware.php.* 3, 145

[3] *http://www.liberouter.org.* 5

[4] Agere, "The challenge for next generation network processors." [Online]. Available: www.agere.com/docs/challenge_new.pdf 10

[5] Alchemy, "Alchemy semiconductor unveils au1000 internet edge processor." [Online]. Available: http://www.thefreelibrary.com/Alchemy+Semiconductor+Unveils+Au1000+Internet+Edge+Processor.-a062704288 14

[6] AMCC, "Product family for packet processors." [Online]. Available: https://www.amcc.com/MyAMCC/jsp/public/browse/controller.jsp?networkLevel=EMBE&superFamily=NETP 15

[7] CISCO, "Parallel express forwarding in the cisco 10000 edge service router." [Online]. Available: http://whitepapers.zdnet.co.uk/0,1000000651,260007268p-39000421q,00.htm 16

[8] EZchip, "Network processor designs for next-generation networking equipment." [Online]. Available: http://whitepapers.silicon.com/0,39024759,60001341p-39000410q,00.htm 20

[9] J. R. A. Jr., B. M. Bass, C. Basso, and R. H. B. et al et al, "Ibm powernp network processor: Hardware, software, and applications." [Online]. Available: http://www.research.ibm.com/journal/rd/472/allen.pdf 21

[10] *Intel® IXP2400/2800 Developer's Tool reference manual.* 28, 154

[11] E. J. Johnson and A. R. Kunze, *Ixp2400-2800 Programming: The Complete Microengine Coding Guide.* Intel Press, 2003. 32

[12] D. E.Comer, "Network systems design using network processors: Intel 2xxx version," 2005. 32

[13] *Snort: Lightweight Intrusion Detection for Networks, http://www.snort.org/.* 37, 38

[14] *Bro: A system for Detecting Network Intruders in Real Time, http://bro-ids.org/*. 37, 38

[15] W. Eatherton and J. Williams, *An encoded version of reg-ex database from cisco systems provided for research purposes*. 37, 38, 51, 58, 67, 79

[16] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proc. of CCS '03*. ACM, pp. 262–271. 38, 39, 136

[17] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *SIGCOMM '06*. 38, 39, 42, 52, 54, 56, 59, 67, 136

[18] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. of ANCS '07*. ACM, pp. 155–164. 38, 40, 45, 79, 136

[19] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. of CoNEXT '07*. ACM, 2007, pp. 1–12. 38, 40, 52, 67, 136

[20] R. Smith, C. Estan, and S. Jha, "Xfas: Fast and compact signature matching," University of Wisconsin, Madison, Tech. Rep., August 2007. 38, 40, 45, 53, 136

[21] ——, "Xfa: Faster signature matching with extended automata," in *IEEE Symposium on Security and Privacy*, May 2008. 38, 40, 45, 53

[22] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," in *SIGCOMM '08*. 38

[23] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975. 39

[24] B. Commentz-Walter, "A string matching algorithm fast on the average," in *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*. London, UK: Springer-Verlag, 1979, pp. 118–132. 39

[25] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," University of Arizona, Tech. Rep. TR-94-17. 39

[26] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," pp. 333–340, 2004. 39, 43, 45

[27] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: Hardware/software ip lookups with incremental updates," 2004. [Online]. Available: citeseer.ist.psu.edu/dittia02tree.html 39, 112, 113, 121

[28] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *ANCS '06*, New York, NY, USA, 2006, pp. 93–102. 39, 79

[29] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. 39

[30] M. Becchi and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,” in *Proc. of ANCS ’07*, 2007, pp. 145–154. 39, 42, 52, 67

[31] M. Becchi and S. Cadambi, “Memory-efficient regular expression search using state merging.” in *Proc. of INFOCOM 2007*, May 2007. 39, 40, 43, 52, 67

[32] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A scalable architecture for high-throughput regular-expression pattern matching,” in *Proc. of ISCA’06*, June 2006. 40, 61

[33] S. Kumar, J. Turner, and J. Williams, “Advanced algorithms for fast and scalable deep packet inspection,” in *ANCS ’06: Proc. of the ACM/IEEE symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2006, pp. 81–92. 40, 103

[34] G. Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. 49, 121

[35] *Intel Network Processors, www.intel.com/design/network/products/npfamily/.* 50, 122

[36] *Michela Becchi, regex tool, http://regex.wustl.edu/.* 52, 59, 67, 81

[37] J. E. Hopcroft and J. D. Ullman, *Introduction To Automata Theory, Languages, And Computation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. 60, 62, 71

[38] M. Becchi and P. Crowley, “Efficient regular expression evaluation: theory to practice,” in *ANCS ’08*, 2008. 61, 82

[39] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 64

[40] G. Khanna, I. Laguna, F. Arshad, and S. Bagchi, “Stateful detection in high throughput distributed systems,” in *IEEE SRDS 2007*. 70

[41] E. Ketcha Ngassam, D. G. Kourie, and B. W. Watson, “On implementation and performance of table-driven DFA-based string processors,” in *Proceedings of the Prague Stringology Conference ’06*, 2006. 74

[42] E. K. Ngassam, B. W. Watson, and D. G. Kourie, “Hardcoding finite state automata processing,” in *SAICSIT ’03*, 2003. 74

[43] K. Ngassam, “Towards cache optimization in finite automata implementations,” Ph.D. dissertation, University of Pretoria, South Africa, 2007. 74

[44] *http://www.circlemud.org/ jelson/software/tcpflow/.* 81

[45] *http://www.intel.com/design/network/products/ npfamily/ixp2800.htm.* 81

[46] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004. 84

[47] M. Nourani and P. Katta, "Bloom filter accelerator for string matching," in *ICCCN 2007: Proceedings of the 16th International Conference on Computer Communications and Networks*, 2007, pp. 185–190. 84

[48] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 254–265, 1998. 84, 176

[49] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, Tech. Rep., 1998. [Online]. Available: citeseer.ist.psu.edu/ptacek98insertion.html 85

[50] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium.* Berkeley, CA, USA: USENIX Association, 2001, pp. 9–9. 85

[51] N. S. Artan and H. J. Chao, "Multi-packet signature detection using prefix bloom filters," in *GLOBECOM 2005: Proceedings of the IEEE Global Telecommunications Conference*, vol. 3, 2005, p. 18111816. 85, 86, 89

[52] G. Varghese, J. A. Fingerhut, and F. Bonomi, "Detecting evasion attacks at high speeds without reassembly," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 327–338, 2006. 85, 86, 89

[53] M. Vutukuru, H. Balakrishnan, and V. Paxson, "Efficient and Robust TCP Stream Normalization," in *2008 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008. 87

[54] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Multilayer compressed counting bloom filters," in *Proc. of INFOCOM '08*, Phoenix, AZ, USA, April 2008. 87, 90, 95, 97

[55] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *Proceedings of ACM SIGCOMM'05*, 2005. 87, 102

[56] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Blooming trees: Space-efficient structures for data representation," in *Communications, 2008. ICC '08. IEEE International Conference on*, May 2008, pp. 5828–5832. 90, 94, 95, 96, 97, 99, 130

[57] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *LNCS 4168, 14th Annual European Symposium on Algorithms*, 2006, pp. 684–695. 90

[58] G. Bianchi, S. Teofili, M. Pomposini, "New directions in privacy-preserving anomaly detection for network traffic," in *Proc. of Network Data Anonymisation (NDA 2008)*, Virginia, USA, October 2008. 94

[59] *http://www.intel.com/design/itanium/documentation.htm.* 94

[60] *http://softwarecommunity.intel.com/isn/Downloads/Intel SSE4 Programming Reference.pdf.* 94

[61] *http://www.intel.com/design/network/products/npfamily/ixp2805.htm.* 94

[62] *http://vincent.amd.com/us-en/assets/content_type/white papers and tech docs/40546.pdf.* 94

[63] *http://www.power.org/resources/reading/PowerISA_V2.05.pdf.* 94

[64] Z. J. Czech, G. Havas, and B. S. Majewski, "Fundamental study - perfect hashing," *Theoretical Computer Science*, vol. 182, no. 1, August 1997. 94, 95

[65] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with 0(1) worst case access time," *J. ACM*, vol. 31, no. 3, 1984. 94

[66] Z. J. Czech, G. Havas, and B. S. Majewski, "An optimal algorithm for generating minimal perfect hash functions," *Information Processing Letters*, vol. 43, no. 5, 1992. 94, 95

[67] R. Pagh, "Hash and displace: Efficient evaluation of minimal perfect hash functions," in *Workshop on Algorithms and Data Structures*, 1999. 94, 95

[68] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1984, vol. 1. 94

[69] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud, "Practical minimal perfect hash functions for large databases," *Commun. ACM*, vol. 35, no. 1, 1992. 94

[70] F. C. Botelho, Y. Kohayakawa, and N. Ziviani, "An approach for minimal perfect hash functions for very large databases," Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, Tech. Rep., 2006. 95

[71] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," *Springer-Verlag Lecture Notes in Computer Science*, vol. 4619, 2007. 95, 101

[72] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect hashing for network applications," in *Proceedings of International Symposium on Information Theory 2006*, 2006. 95, 99, 101

[73] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "Hexa: Compact data structures for faster packer processing," in *Proc. of ICNP 07*, 2007. 103

[74] D. Ficara, S. Giordano, S. Kumar, and B. Lynch, "Divide and discriminate: Algorithm for fast and deterministich hash lookups," in *ANCS '09: Proc. of the ACM/IEEE symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2009. 103

[75] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, and O. Boy, "The bloomier filter: An efficient data structure for static support lookup tables," in *Proc. of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004, pp. 30–39. 103, 108

[76] P. Hall, "On representatives of subsets," *J. London Math. Soc.*, vol. 10, pp. 26–30, 1936. 106

[77] R. Motwani, "Average-case analysis of algorithms for matchings and related problems," *Journal of the ACM*, vol. 41, pp. 1329–1356, 1994. 107

[78] J. Hopcroft and R. Karp, "An n5/2 algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2, 1973. 107

[79] M. Mitchell, "An introduction to genetic algorithms," 1996. 107, 108

[80] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. of the ACM SIGCOMM '97*. New York, NY, USA: ACM, 1997, pp. 3–14. 112, 113, 121

[81] P. Gupta and N. Mckeown, "Packet classification using hierarchical intelligent cuttings," in *in Hot Interconnects VII*, 1999, pp. 34–41. 112, 136

[82] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *SIGCOMM '03*. 112, 136

[83] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *SIGCOMM*, 1998, pp. 203–214. [Online]. Available: citeseer.ist.psu.edu/lakshman98highspeed.html 112

[84] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM*, 1999, pp. 135–146. [Online]. Available: citeseer.ist.psu.edu/article/srinivasan99packet.html 112

[85] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *INFOCOM '98*. 114

[86] N.-F. Huang and S.-M. Zhao, "A novel ip-routing lookup scheme and hardware architecture for multigigabit switching routers," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, June 1999. 114

[87] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest prefix matching using bloom filters," 2003. [Online]. Available: citeseer.ist.psu.edu/dharmapurikar00longest.html 114, 122, 130

[88] *Route Views 6447, http://www.routeviews.org/*. 114, 122, 130, 134

[89] V. Srinivasan and G. Varghese, “Faster ip lookups using controlled prefix expansion,” *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 1–10, 1998. 117

[90] *Classbench, A Packet Classification Benchmark, http://www.arl.wustl.edu/classbench/.* 118

[91] *BGP potaroo, http://bgp.potaroo.net.* 121

[92] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, “Scalable high speed ip routing lookups,” *SIGCOMM Comput. Commun. Rev.*, 1997. 122

[93] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* McGraw-Hill Book Company, 2001. 125, 126

[94] K. S. Kim and S. Sahni, “Ip lookup by binary search on prefix length,” in *ISCC ’03.* Washington, DC, USA: IEEE Computer Society, p. 77. 125

[95] *http://www.spirent.com.* 134, 165

[96] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, “Router plugins: A software architecture for next generation routers,” in *IEEE/ACM transactions on Networking*, 1998, pp. 229–240. 135

[97] G. V. L. Qiu and S. Suri, “Fast firewall implementations for software and hardware-based routers,” *Proc. of the 9th International Conference on Network Protocols (ICNP)*, 2001. 135

[98] S. S. P. Warkhede and G. Varghese., “Fast packet classification for two-dimensional conflict-free filters,” *Proc. IEEE Infocom*, 2001. 135

[99] H. Adisheshu, “Services for next-generation routers,” *Ph.D. dissertation, Washington University Computer Science Department*, 1998. 136

[100] T. Woo, “A modular approach to packet classification: Algorithms and results,” *Proc. IEEE Infocom*, 2000. 136

[101] E. Cohen and C. Lund, “Packet classification in large isps: design and evaluation of decision tree classifiers,” in *Sigmetrics*, 2005, pp. 73–84. 136

[102] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, “Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough,” *Sigmetrics Perform. Eval. Rev.*, vol. 35, no. 1, 2007. 136

[103] H. Song, J. Turner, and S. Dharmapurikar, “Packet classification using coarse-grained tuple spaces,” in *ANCS ’06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems.* New York, NY, USA: ACM, 2006, pp. 41–50. 136

[104] *Lawrence Berkeley National Labs, tcpdump/libpcap, Network Research Group, http://www.tcpdump.org.* 144

[105] *Wireshark protocol Analyzer (was Ethereal), http://www.wireshark.org.* 144

[106] *Ntop network traffic probe, http://www.ntop.org.* 144

[107] L. Deri, "Improving passive packet capture: Beyond device polling," in *SANE 2004*. 144

[108] ——, "Passively monitoring networks at gigabit speeds using commodity hardware and open source software," in *PAM 2003*. 144

[109] *Endace, http://www.endace.com.* 144

[110] T. Wolf, R. Ramaswamy, S. Bunga, and N. Yang, "An architecture for distributed real-time passive network measurement," in *MASCOTS 2006*. 145

[111] A. D. Pietro, D. Ficara, S. Giordano, F. Oppedisano, G. Procissi, and F. Vitucci, "A network processor based architecture for multi gigabit traffic analysis," in *International Journal of Communication Systems, 2009*. 145

[112] *SCAMPI project, http://www.ist-scampi.org.* 145

[113] L. Deri, "ncap: Wire-speed packet capture and transmission," in *End-to-End Monitoring, 2005*. 145

[114] P. Saul, "Direct digital synthesis," in *Circuits and systems tutorials, 1996*. 147

[115] S. F. Donnely, "High precision timing in passive measurements of data networks," in *PhD Thesis, 2002*. 147

[116] *Xilinx, http://www.xilinx.com.* 148

[117] *http://www.isi.edu/nsnam/ns/.* 151

[118] N. Bonelli, S. Giordano, G. Procissi, and R. Secchi, "Brute: A high performance and extensibile traffic generator," in *Proc. of Int'l Symposium on Performance of Telecommunication Systems (SPECTS'05)*, July 2005. 151, 152, 153

[119] *http://caia.swin.edu.au/genius/tools/kute/.* 151, 152

[120] *http://rude.sourceforge.net/.* 151, 152, 153

[121] *http://mgen.pf.itd.nrl.navy.mil/.* 151, 152

[122] *http://protocols.netlab.uky.edu/ esp/pktgen/.* 152, 154

[123] R. Bolla, R. Bruschi, M. Canini, and M. Repetto, *A High Performance IP Traffic Generation Tool Based On The Intel IXP2400 Network Processor*, ser. Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements. Springer Berlin Heidelberg, 2006, pp. 127–142. 152, 154

[124] A. Botta, A. Dainotti, and A. Pescape, "Multi-protocol and multi-platform traffic generation and measurement," in *Proc. of INFOCOM 2007 DEMO Session*, May 2007. 152, 153

[125] S. Avallone, A. Pescape, and G. Ventre, “Analysis and experimentation of internet traffic generator,” in *Proc. of New2an 2004, International Conference on Next Generation Teletraffic and Wired/Wireless Advanced Networking*, February 2004. 153

[126] *http://netgroup-serv.iet.unipi.it/brute/.* 153

[127] A. Abdo, H. Awad, S. Paredes, and T. J. Hall, “Oc-48 configurable ip traffic generator with dwdm capability,” in *Proc. of the Canadian Conference on Electrical and Computer Engineering*, May 2006, pp. 1842 – 1845. 154

[128] *Intel® IXP2800 Hardware reference manual.* 155

[129] *Intel Corporation, 21555 Non-Transparent PCI-to-PCI Bridge User’s manual.* 158

[130] *Linux Device Drivers, Third Edition, http://lwn.net/Kernel/LDD3/.* 159

[131] K. Park and W. Willinger, *Self-Similar Network Traffic: An Overview.* Wiley Interscience, 1999. 167

[132] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commu. of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970. 175

[133] A. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” *Internet Mathematics*, vol. 1, no. 4, 2005. [Online]. Available: http://www.internetmathematics.org/volumes/1/4/Broder.pdf 175