Parallel and Distributed Processing in High Speed Traffic Monitoring

Mihai-Lucian Cristea

Parallel and Distributed Processing in High Speed Traffic Monitoring

Proefschrift

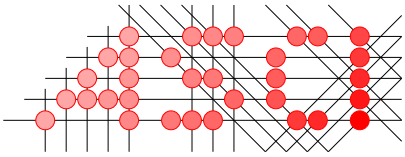
ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. P. F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 1 oktober 2008
klokke 15:00 uur

door

Mihai-Lucian Cristea
geboren te Galaţi, România
in 1976

Samenstelling promotiecommissie:

promotor:	Prof.dr. H.A.G. Wijshoff	Universiteit Leiden
co-promotor:	Dr. H.J. Bos	Vrije Universiteit
referent:	Dr. E.P. Markatos	FORTH, Greece
overige leden:	Prof.dr. E. DePrettere	Universiteit Leiden
	Prof.dr. F.J. Peters	Universiteit Leiden
	Prof.dr. J.N. Kok	Universiteit Leiden
	Dr. C.Th.A.M. de Laat	Universiteit van Amsterdam
	Dr. B.H.H. Juurlink	Technische Universiteit Delft



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number: 164.

Parallel and Distributed Processing in High Speed Traffic Monitoring
Mihai-Lucian Cristea.
Thesis Universiteit Leiden. - With ref. - With summary in Dutch
ISBN 978-973-1937-03-8

Printed in Romania by ALMA PRINT Galaţi

Contents

Chapter 1

Introduction

How could anybody imagine in the late 1960s, when the Internet was born on those 50 Kbps wires of the ARPANET, that nowadays, less than five decades later, we are using connection speeds of multiple Gbps over millions of nodes. Although the Internet development started as a military project, ARPANET, it went to public domain and is currently supported world-wide from people's need to share and exchange their knowledge. Currently, there is a huge amount of information available on the Internet that no one man can assimilate by himself in a lifetime. Extrapolating from these facts, can anybody imagine how fast our machines will communicate in the next few decades? What kind of material or immaterial support will be used by that time for information exchange? How will the 'network of networks' (the Internet) manage/control itself?

Since the Internet was born on the first packet-switched data networks, there has been a need for traffic processing at the nodes of this network (e.g., for routing). Currently, however, the need arises for increasingly complex processing at these nodes. For instance, increasing concerns about security demand efficient scanning of payloads and other needs such as network monitoring. Unfortunately, it seems that the network speed increases at least as fast as the processing capacity and probably faster [1]. It has often been suggested that parallelism is the only way to handle future link rates [2–4]. In this thesis, the assumption is that link rates grow so fast as to exceed the capacity of a single processor. The question we try to address is how to handle the link rate of the future. To limit the scope of this research to a manageable problem domain, we restrict ourselves primarily to network monitoring applications (although we also consider other domains).

This chapter briefly summarises the research questions and the approach towards a solution in the first two sections, followed by the current state-of-the-art in traffic monitoring which will be described in Section 1.3. The thesis statement and argumentation are presented in Section 1.4. And finally, an overview of the main results is shown in Section 1.5.

1.1 Problem definition

In recent years there has been a growing gap between the link speeds and the processing and storage capacity for user traffic that flows through the Internet links. On the one hand, the link speed improvements are given mostly by the advances in optical fields. On the other hand, the evolution of processing capacity is much slower than that of link speeds because it is limited by the (slow) evolution of memory access time technology. Using fast memory like SRAM instead of DRAM to cope with the memory bottleneck is a feasible solution for many applications in the computer industry where caching can be aggressively exploited because the applications exhibit a lot of locality of reference. Unfortunately, most networking applications offer very little locality of reference, so the benefit of caching are small. Figure 1.1 illustrates in logarithmic scale, as observed by McKeown [5], a normalised growth of the trends of aggregated user traffic that passes the routers of a typical backbone in the Internet ① versus DRAM memory access time ②.

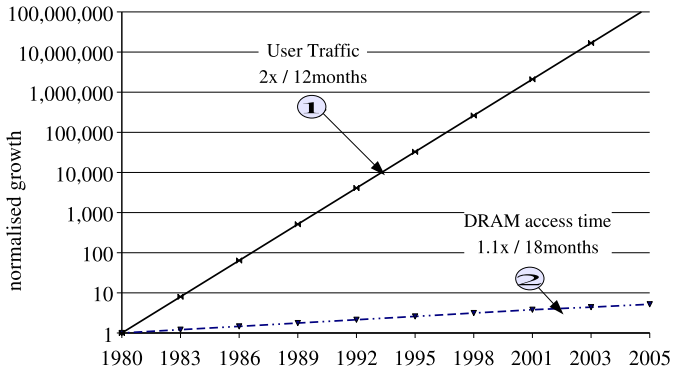


Figure 1.1: Trends in technology and traffic.

The memory bottleneck pushes us towards using parallel and distributed systems. Modern networking equipment often uses distributed systems made of tightly coupled parallel systems: multi-core. However, when using multi-cores hardware systems we need to design and develop networking applications that map onto the parallel system. Moreover, when using different hardware systems for various traffic processing tasks (e.g., a commodity PC for traffic shaping or audio/video streams processing specialised hardware) we need to address a heterogeneous distributed processing system. How can we build such a heterogeneous distributed system for traffic processing?

In addition to the distributed demands, traffic processing systems also face the problem of changing conditions. For instance, a traffic processing application may be designed and developed for certain traffic assumptions. However, one day, the traffic exceeds the assumptions made at design time (e.g., increase in service users, introduction of a new popular service, or malicious traffic). Therefore, a second demand is to build a 'distributed traffic processing system' that can autonomously adapt to an unforeseen and changing environment. For this purpose, we need to address the following research questions:

- initial system state (at deployment time): how to compute a proper mapping of the

system requirements onto a given distributed hardware architecture composed of heterogeneous nodes;

- continuous adaptive system (at runtime): how to adjust the distributed processing system (e.g., by traffic re-routing or tasks re-mapping) according to the environment changes so that the entire system remains stable and works at an optimal level.

1.2 Approach

In these days, there is an increasingly demand for traffic monitoring at backbone speeds. Most of the advanced traffic monitoring systems such as network intrusion detection systems (NIDS) need to process all incoming packets by touching the entire packet byte stream in order to scan for virus signatures, for instance. Moreover, other traffic monitoring systems such as network intrusion prevention systems (NIPS) need, in addition to the NIDS, to perform on each packet intensive computations like checksums, packet fragmentation for transmission, encryption/decryption, etc.

Current, programmable traffic processing systems build on one (or both) of the following available hardware processing architectures: a general purpose CPU such as Intel Pentium 4, or specifically designed embedded systems for packet processing such as Intel IXP2xxx network processors. Both architectures present the same major barrier that stands against providing the processing power required by the traffic monitoring applications: the memory bottleneck. The processing technology is trying hard to cope with the memory bottleneck by using parallel threads or cores and multiple memory buses so as to hide the memory latency.

Assuming an average Internet packet size of 250Bytes, Figure 1.2 illustrates the packet inter-arrival time for current and future link speeds. In other words, a traffic processing system has to receive, process, and eventually transmit every incoming packet within a certain time determined by the packet inter-arrival time so as to keep up with the link speed in use. When the time that the processing system spends on each packet exceeds this packet inter-arrival time, then the system starts dropping packets because a new packet arrives before the system finished with the current one.

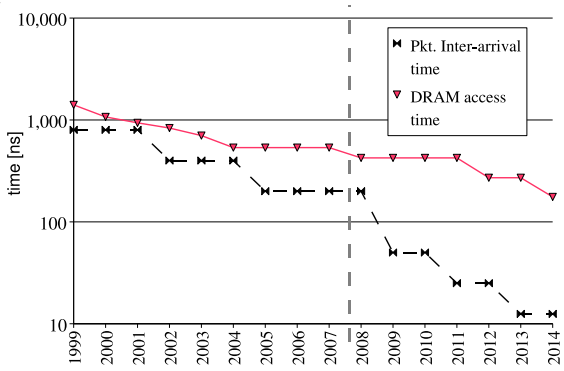


Figure 1.2: Link versus memory access speeds.

The packet inter-arrival time chart shown in Figure 1.2 decreases exponentially with the use of higher link speeds in time. Although the latest official link speed is not in use currently, the first optical transceiver working at OC-3072 (cca. 160 Gbps) speed was already announced by IBM on 29 March 2007. Figure 1.2 also illustrates the memory access time needed to write and read a chunk of 250 data bytes in an external DRAM memory at different moments in time, as the memory technology evolved and is predicted to evolve by Inphi Corporation [6]. However, when comparing the evolution of the packet inter-arrival time against those of the memory access time, we can see that there is a growing disparity. Moreover, this disparity does not include the time a general CPU architecture would need to transfer the data across other buses such as the PCI express.

'Parallelism' is a well known concept that helps to build architectures that cope with the disparity risen because of a bottleneck in the data process flow. Since the end of the 90s, network processor architectures specialised in packet processing were built on a 'tightly coupled' parallelism concept: multiple cores on the same silicon. The external shared memory (DRAM) was connected to the multi-core architecture through parallel buses so as to hide the memory latency as much as possible. However, as shown in Figure 1.2, after a certain link speed the disparity becomes so large that even a single parallel processing system is not sufficient and we will show how we can cope with higher speeds by distributing the workload onto a distributed architecture.

In our context of traffic processing, such a system works as follows: First, there is a *traffic splitter* that splits the input traffic in substreams and distributes them on a hierarchy of possible heterogeneous *processing nodes*. Next, each processing node performs, in parallel, specific traffic processing tasks over its received stream. In the end, the processed packets can be modified and sent out, dropped, or forwarded out as they came in. These actions belong to the packet processing tasks deployed by a user on the entire distributed system. In addition to the processing actions, a supervisor host can collect some of the processing results (e.g., packet counters, virus occurrence numbers) from all processing nodes. These results are then available for further investigation by the user (e.g., a system administrator).

Although the idea of a distributed network monitor was first advocated by Kruegel *et al.* in May 2002 [7], it was used only in the context of a distributed intrusion detection system development using one traffic splitter and several other hosts for distributed processing. Another usage of the traffic splitting idea was introduced by Charitakis *et al.* in [8] also in a NIDS. Although their implementation used a heterogeneous architecture (a network processor card plugged in a commodity PC), the splitter was used to separate the traffic processing of an NIDS into two steps: 'early filtering' and 'heavy processing'. The first step was designed to process a few bytes (e.g., packet header fields) so as to slice and load balance the traffic onto a distributed NIDS sensors. While the 'early filtering' step was implemented on a network processor card, the 'heavy processing' step was implemented in several commodity PCs using the `Snort` NIDS software. Our approach uses the splitter idea in a generalised context of a fully distributed processing system composed of processing nodes that form a processing hierarchy where nodes near the root perform traffic splitting and processing together, and the nodes near the leaves perform only 'heavy processing' tasks. Our architecture is fully programmable by the way the traffic is split/routed across the distributed nodes and is designed to support various traffic processing applications and aims to cope with speed, scalability, and heterogeneous demands.

The approach proposed in this thesis is illustrated in Figure 1.3 and consists mainly of a distributed traffic processing system that *processes* the input traffic, sends out some of the processed traffic, and *interacts* with a user. In addition, the illustrated architecture provides autonomy and self-adaptive properties by using a 'supervisor' component that *monitors* the system and environment states and *re-configures* the system so the entire architecture remains stable regardless of the environment changes (e.g., increase in traffic input, hardware failure).

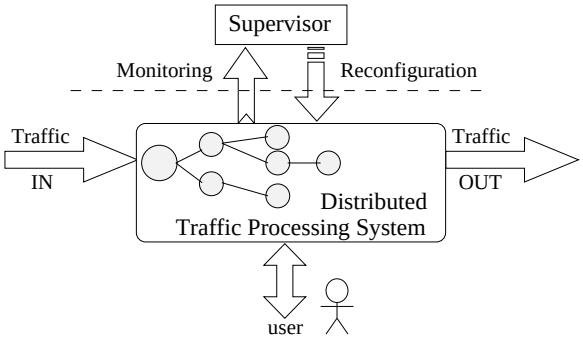


Figure 1.3: Self-adaptive architecture for traffic processing.

We have chosen to use a distributed system for the purpose of traffic processing for the following reasons. First, there is an increasing demand for intensive traffic processing applications such as the Network Intrusion Detection Systems (NIDS) and the Network Intrusion Prevention Systems (NIPS). Second, running the traffic processing applications at backbone line rate (10 Gbps and more) is a challenge in itself. Third, special hardware systems called network processors (NPs) provide hardware parallelism (using multi-cores) to cope with high bandwidth from a backbone line, but programming them is difficult and the performance is still insufficient. For instance, even a basic NIDS implementation on a network processor (as Bos *et al.* show in [9]) cannot perform more than 1 Gbps. Fourth, the life cycle of the special purpose hardware for traffic processing (e.g., network processors) gets shorter and shorter. Moreover, these hardware systems are expensive and a customer often has multiple hardware system generations in use. We see an increasing demand for using heterogeneous hardware for processing nodes in a distributed system. Fifth, various traffic processing applications make use of hardware designed for a specific purpose: network processors for generic traffic processing such as routing, monitoring, limited payload scanning; custom FPGA designs for specific processing tasks such as processing video streams; etc. Using a distributed system is a solution for unifying every specific traffic processing node. Concluding, a distributed system offers the following benefits: (1) heterogeneity (using a federated architecture composed of various hardware processing system such as commodity PCs, network processors); (2) scalability (practical because there is no limitation for the size of the system). At the same time, a distributed approach comes at a price: the increased complexity of the system.

The traffic processing system in which we are interested is designed to work at the network edge of enterprise gateways. Once the system is deployed by an administrator, we expect it to work non-stop. However, it is well known that there are always peaks in the network traffic (e.g., rush hours or malicious traffic). Therefore, in addition to the distributed

property we also propose to use an automatic control system (the 'supervisor' in Figure 1.3) that provides (3) a high availability property, meaning robustness to congestions for unexpected environment changes.

1.3 State of the Art in network traffic monitoring

In the following subsections we describe the state-of-the-art problems in traffic monitoring and the current related technologies.

1.3.1 Network monitoring applications

The explosion of network based applications increases the complexity of the traffic by means of more and more protocols and their relationships to be analysed by any system administrator in case of network problems. Every network user produces and consumes traffic data through the applications he/she uses. For example, in the early days of Internet, most of the traffic seen at a corporate gateway was produced by e-mail client/server applications and could be monitored easily by a commodity PC over a 10/100 Mbps connection. At the present time, the peer-to-peer and web-service applications have risen the traffic bandwidth to such a level that soon no piece of hardware could monitor, alone, the entire traffic that passes a corporate backbone. Moreover, there are specific demands for handling high speed traffic (multi Gbps) in various research fields such as super computing, grid computing, etc. For instance, a scientist needs to access, remotely, the results of certain physics experiments from his office across the ocean. This action requests huge amount of data over a (physically) long connection that bypasses the regular Internet and uses specially demanded fibre optic links. In order to provide such links on-demand dynamically, we need to monitor the traffic at these high speeds so as to prevent un-authorised traffic using the requested links.

For a convenient description of the network tools, we classify them in two categories: high-level and low-level. The latter are run-time libraries and tools for packet processing at the lowest level (OS kernel or specialised hardware) on top of which the former are built: network monitoring applications.

High-level: applications

Although there are monitoring applications that need to analyse the full backbone traffic such as network intrusion detection or prevention systems, there are also monitoring applications that are interested in only part of the traffic. Such tools that monitor only part of the traffic are: network debugging that answer to questions like 'what is the performance between two clients?', traffic accounting tools that identify 'what traffic is p2p', etc.

Traffic monitoring applications can diagnose network problems by protocol analysis. For instance, a performance analysis of real-time streams between two nodes may take into account the following parameters:

- *total delay*: it is given by the network latency and usually is a stable value for any connection;

- *bandwidth consumption*: it may be shown by the average, or maximum value or, the peak burst value of the moving average bit rate;
- *jitter*: distribution of packet inter-arrival time determines how many packets are delayed beyond the jitter buffer;
- *packet loss*: determine the effect on quality due to packet loss for real-time streams.

Traffic monitoring provides critical information for internet service providers (ISPs) looking to optimise network speed, content delivery and performance of subscribed services on the network. Advanced reporting also helps ISPs categorise unidentified traffic for increased security from unidentified P2P traffic, for instance.

In the end, we see an increasing demand for a monitoring system that supports multiple monitoring applications processing together the incoming traffic within the same hardware due to speed demands and cost reasons.

For example, multiple monitoring applications (e.g., `snort` [10], `tcpdump` [11], `ntop` [12], `CoralReef` [13]) access identical or overlapping sets of packets. Therefore, we need techniques to avoid copying a packet from the network device to each application that needs it. For instance, the FFPF tool [14] provides different 'copy' semantics so as to minimise the packet copying. The most common mechanism is using shared buffers. As an example, the FFPF implementation uses a large shared packet buffer (where all received packets are stored) and one small index buffer for each application. Applications use the local indexes to find the interesting packets in the shared buffer. One of the copy semantic features available in FFPF is called 'copy-once'. In other words, the system copies only once the received packet from NIC to the host's shared memory. Usually this copy is transparently performed by the network card itself through DMA channels. Another copy semantic provided by FFPF is 'zero-copy'. In zero-copy, every incoming packet is stored in a large shared buffer within the network card and the packet index is provided to each monitoring application running locally in the card or remotely in the host. Then the application uses this packet index to point to the main packet data from the card's buffer and checks the fields it is interested in. There is a trade-off between the usage of zero-copy or copy-once, depending on the application demands and on the available hardware support. For instance, when we have monitoring applications that are interested in processing the entire stream of incoming packets (e.g., NIDS) it is feasible to use copy-once. However, using zero-copy may be better when the most processing takes place on the card and only statistics are sent to the host, or when the host accesses packets irregularly and only for checking a few bytes of the packet (e.g., various protocol performance analysers).

One of the most intensive traffic processing applications is network intrusion detection/prevention systems because they usually process the full traffic (entire packet payload). In Information Security, intrusion detection is "the act of detecting actions that attempt to compromise the confidentiality, integrity or availability of a resource" [15]. Intrusion detection does not, in general, include prevention of intrusions. In other words, a network intrusion detection system (NIDS) detects computer systems threats by means of malicious usage of the network traffic. A NIDS works by identifying patterns of traffic presumed to be malicious that cannot be detected by conventional firewalls such as network attacks against vulnerable services, data driven attacks on applications, etc. A network intrusion prevention system (NIPS) works differently from IDS in the sense that it analysis each packet for malicious

content before forwarding it and drops packets sent by an intruder. To do so, the IPS has to be physically integrated into the network and needs to process the actual packets that run through it, instead of processing copies of the packets at some place outside the network. Therefore, independent of the way they are built, all IPSes introduce the same problem: a decrease in performance (e.g., inter-packet delay) of the network they try to protect.

A lightweight network intrusion detection system is `Snort` [10]. Snort is the most widely deployed intrusion detection and prevention technology worldwide and has become the de facto standard for the industry. Snort is an open-source software capable of performing real-time traffic analysis and packet logging on IP networks. It can perform protocol analysis, content searching/matching and can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, OS fingerprinting attempts, and much more. Snort uses a flexible rules language to describe traffic that it should collect or pass, as well as a detection engine that uses a modular plugin architecture. Snort has a real-time alerting capability as well as alerting mechanisms for syslog, or a user specified file. However, Snort copes fairly well with low speed links, but it cannot handle multi-gigabit link rates.

Other intensive traffic processing applications such as *transcoders* are currently adjusting the encoded video streams (e.g., JPEG-2000, MPEG-4) to the unreliable transmission medium which uses fast and low transmission-delay protocols such as UDP. On the one hand, the encoded video streams are sensitive to the arrival order of consequent packets which compose video frames. On the other hand, a best effort protocol such as UDP does not offer any guarantee of the packet arrival correctness. Therefore, the encoded multimedia stream needs to be 'transcoded' into a more appropriate format stream for the transmission capabilities. For example, a transcoding from the layered representation in which any frame loss brings a huge quality drop (e.g., video de-synchronisation) to multiple distinct descriptions of the source which can be synergistically combined at the other end would enhance the quality [16]. Transcoding may also use algorithms such as Forward Error Correction (FEC) which requires intensive computation in real-time and thus, we see the need for parallel and distributed architectures in order to provide the required processing power for the current and the future encoded media streams.

Other intensive traffic processing applications are those that require encryption computation. For example, network oriented applications which require specific networks such as direct end-to-end fibre interconnection (e.g., grid computation, corporate database transfers) need to provision a safe authentication for the dynamically demanded fibres. Therefore, encryption per-packet is sometimes used to provide the end-to-end paths over multiple network domains where different policy applies [17]. Again, parallel and distributed architectures help to bring the required traffic processing power.

Low-level: run-time libraries

Most of the high-level applications are built on top of a few, common, low-level libraries or tools. In this section, we briefly describe the evolution of the packet filtering tools and their techniques.

The Berkeley Packet Filter (`BPF`) [18] was one of the first generation of packet filtering mechanism. BPF used a virtual machine built on a register-based assembly language and a directed control flow graph. However, BPF lacks in an efficient filter composition technique because the time required to filter packets grows linearly with the number of con-

catenated filters and hence, BPF does not scale well with the number of active consumers. MPF [19] enhances the BPF virtual machine with new instructions for demultiplexing to multiple applications and merges filters that have the same prefix. This approach is generalised by PathFinder [20] which represents different filters as predicates of which common prefixes are removed. PathFinder is interesting in that it is well suited for hardware implementation. DPF [21] extends the PathFinder model by introducing dynamic code generation. BPF+ [22] shows how an intermediate static single assignment representation of BPF can be optimised, and how just-in-time-compilation can be used to produce efficient native filtering code. Like DPF, the Windmill protocol filters [23] also targets high performance by compiling filters in native code. And like MPF, Windmill explicitly supports multiple applications with overlapping filters. All of these approaches target filter optimisation especially in the presence of many filters. Nprobe [24] aims at monitoring multiple protocols and is therefore, like Windmill, geared towards applying protocol stacks. Also, Nprobe focuses on disk bandwidth limitations and for this reason captures as few bytes of the packets as possible. In contrast, FFPF [14] has no *a priori* notion of protocol stacks and supports full packet processing.

The current packet filtering generation, so called packet classification, uses advanced searching algorithms for rules matching. Packet classification is currently used in many traffic processing tools such as traffic monitoring, routers, firewalls, etc. While in the early days of packet filtering, tools like iptables used a matching algorithm which traverses the rules in a chain linearly per packet. However, nowadays, when many rules have to be checked, a linear algorithm is not suitable anymore. There are many solutions that may use, for instance, geometric algorithms, heuristic algorithms, or specific hardware accelerated searching algorithms. For instance, nf-HiPAC [25] offers a packet classification framework compatible with the known iptables. An ongoing research effort in a packet classification using regular expressions is described in [26, 27].

We note that the most popular high-level tools in use currently (e.g., tcpdump, ntop, snort) are built on top of the pcap library (libpcap) [28]. libpcap provides a cross-platform framework for low-level network monitoring. We notice that libpcap uses a packet filtering mechanism based on the BSD packet filter (BPF).

1.3.2 Specialised hardware for network monitoring

The current demands of monitoring high speed network links such as backbone links and the increasing speed of those links beyond the processing capabilities of nowadays' commodity PCs opened the market for specialised architectures for traffic processing. Such architectures uses specialised hardware systems that accelerate specific packet processing functions (e.g., hashing, encryption, searching algorithms) similarly to the graphic accelerators in the 1990s. Currently, the available hardware technology for traffic processing, namely network processors, make use of parallelism to cope with the increasing gap between the link and the memory access speeds.

Filtering and processing in network cards was initially promoted by some Juniper routers [29] and the Scout project [30]. In programming Intel IXP network processors (IXPs), the most notable projects are NPClick [31] and netbind [32]. Although NPClick and netbind introduce interesting programming models, they were not designed for monitoring.

Although the network processors are the current state-of-the-art of specialised technology

for traffic processing, they are surpassed by FPGA technology for specific intensive computation such as custom encryption algorithms, or other mathematically based encoders/decoders. FPGAs are fully programmable chips and suitable for experimental software development in the hardware until an ASIC chip may be deployed.

SpeedRouter [33] is one of the first FPGA-based network processor, but as such, it may require more user programming than most of the network processors (NPs). While the device provides basic packet-forwarding functions, it is up to the user to program functions such as key extraction and packet modification. SpeedRouter was designed for a cut-through data path, as opposed to the store-and-forward nature of most pipelined architectures met in NPs.

DAG cards [34] are another example of NICs based on FPGA technology. They are designed especially for traffic capture at full link rate. DAG cards make use of FPGA parallelism to be able to process the incoming traffic at high speeds, they offer accurate time-stamp using GPS signals (7.5ns resolution), and transfer the full traffic directly to the host PC's memory for a further processing/storing.

The value seen in FPGA prototyping for emulating ASIC designs is recognised across a growing spectrum of design applications. For instance, Shunt [35] is an FPGA-based prototype for an Intrusion Prevention System (IPS) accelerator. The Shunt maintains several large state tables indexed by packet header fields, including IP/TCP flags, source and destination IP addresses, and connection tuples. The tables yield decision values that the element makes on a packet-by-packet basis: forward the packet, drop it, or divert it through the IPS. By manipulating table entries, the IPS can specify the traffic it wishes to examine, directly block malicious traffic, and 'cutting through' traffic streams once it has had an opportunity to 'vet' them, all on a fine-grained basis.

Using reconfigurable hardware for increased packet processing efficiency was previously explored in [36] and [37]. Our architecture differs in that it provides explicit language support for this purpose. As shown in [38], it is efficient to use a source-to-source compiler from a generic language (Snort Intrusion Detection System) to a back-end language supported by the targeted hardware compiler (e.g., Intel μEngineC, PowerPC C, VHDL). We propose a more flexible and easy to use language as front-end for users. Moreover, our FPL language is designed and implemented for heterogeneous targets in a multi-level system.

Offloading the traffic handling down to the NIC

The current hardware technology contributes to improve the way the traffic is processed in a commodity PC by offloading intensive computations from general purpose CPU down to the NIC card. For instance, looking at the workload that a general purpose CPU spends on processing the network traffic in any user PC connected to a network, it is known that the TCP protocol requires the most CPU time spent on traffic handling before the data arrives at the application. Hence, the first 'protocol accelerators' arrived on the market are: TCP Offload Engine (TOE) and TCP segmentation offloading (TSO).

TCP Offload Engine (TOE) is the name for moving part or all of the TCP/IP protocol processing down to a specialised NIC. In other words, by moving of the TCP/IP processing to a separate dedicated controller off the main host CPU, the overall system TCP/IP performance can be improved. Currently, the developer community is reluctant to adopt TOE, as we will see shortly.

In addition to the protocol overhead that TOE addresses (e.g., sliding window calculations

for packet acknowledgment and congestion control, checksum and sequence number calculations), it can also address some architectural issues that affect many host based endpoints. For example, a TOE solution, located on the network interface, is placed on the other side of the (slow) PCI bus from the CPU host so it can address some I/O efficiency issues. Thus the data needed to cross the TCP connection can be sent to the TOE from the CPU across the PCI bus using large data burst sizes with none of the smaller TCP packets having to traverse the PCI bus.

On the other hand, there are several reasons to consider the full network stack offload (TOE) of little use, as also observed by J. Mogul [39] and summarised as follows:

- *Security updates:* A TOE network stack is proprietary source firmware. System administrators have no way to fix security issues that arise;
- *Hardware-specific limits:* TOE NICs are more resource limited than the overall computer systems. This is most readily apparent under load factors which arise when trying to support thousands of simultaneous connections. TOE NICs often simply do not have the memory resources to buffer thousands of connections;
- *Dependency on vendor-specific tools:* In order to configure a TOE NIC, hardware-specific tools are usually required. This dramatically increases support costs.

The TCP segmentation offloading (TSO) features can improve performance by offloading some TCP segmentation work to the adapter and cutting back slightly on bus bandwidth. For instance, when large chunks of data are to be sent over a computer network, they need to be first broken down into smaller segments that can pass through all the network elements like routers and switches between the source and destination computers. This process is referred to as segmentation. Segmentation is often done by the TCP protocol in the host computer. Offloading this work to the network card is called TCP segmentation offload (TSO).

However, despite its benefits, TSO has also several drawbacks. One drawback is that once a TSO capability was developed to work in an OS vendor's software, the NIC hardware and driver modifications that allow offloading will not work with other OSes. A second drawback is that the offload capability is one-way (transmit) only. A third drawback is that the host cannot offload a data block larger than the remote endpoint's receive window size.

1.3.3 Storing traffic at high-speeds

Traffic monitoring by means of traffic capture requires also storing functionality. The received traffic from a network tap needs to be transferred to disks for a further detailed analysis. Support for high-speed traffic capture is provided by OCxMon [40]. Like the work conducted at Sprint [41], OCxMon supports DAG cards to cater to multi-gigabit speeds [34]. Besides using a host PC for traffic capture from a high speed link onto a storage disk, another research area is to use custom FPGA designs as a glue between fibre optic for traffic input and fast serial data lines (e.g., SATA) for output to storage disks. Such systems are currently in use by Storage Area Network (SANs) systems [42].

1.4 Traffic processing at high link rates

In this section we present the speed problems that arises when more and more tools want to process packets at higher and higher speeds as shown in the previous sections.

Figure 1.4 shows, in logarithmic scale, the trends in the technologies related to networking as found by McKeown [5]. We see the smallest technology improvement in the DRAM memory access time (chart ①). We can also see that according to Moore's Law, processor complexity (and hence the performance) is constantly growing at mostly the same rate as the line capacity until year 1995 (chart ②). By that time, the optical revolution in wave division multiplexing (DWDM) came together with another prophecy: Gilder's law [1]. George Gilder predicted that "network bandwidth would triple every year for the next 25 years" (chart ③). The most rising chart, user traffic (chart ④), is constantly growing since the early days of the Internet, and sustained by the web innovation in beginning of 1990s, by doubling each year.

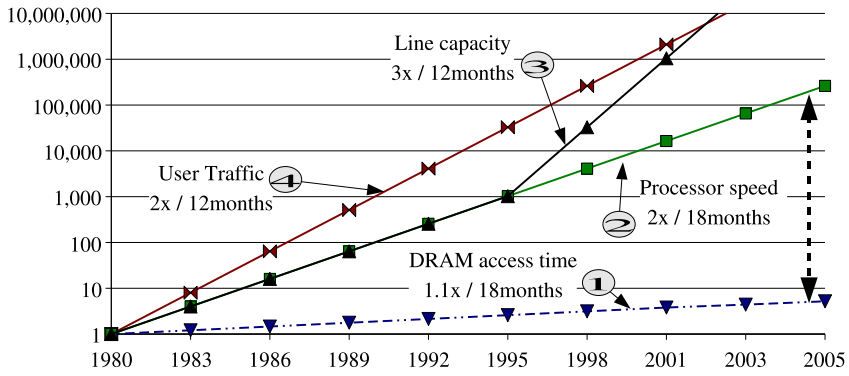


Figure 1.4: Trends in technology, routers and traffic.

Considering the disparity between memory access time (chart ①) and processors speed (chart ②), we can say that accessing memory becomes twice as expensive every 18 months [5]. In this context, a generic CPU uses bigger caches and better pre-fetching algorithms, and networking uses more processing cores in order to hide the memory latency because networking cannot offer the locality of reference of the processed data on which a caching technology needs to work.

For example, as mentioned earlier, the networking industry may use specifically designed processors called network processors (NPs). NPs use a highly parallelised internal architecture (typically more than 8 cores), several memory types (e.g., SRAM, DRAM) organised in multiple banks and interconnected through parallel buses. NPs use multiple cores specialised in packet processing in order to offer flexibility (they support many applications and deal with protocol and standards changes), and reduce risks (bugs are easier to fix in software than in hardware). NPs try hard to hide memory latency by means of asynchronous memory access because conventional caching is not suitable for networking applications (no temporal or spatial locality, cache misses decrease throughput).

Summarising, the bottleneck moved from processor clock speeds down to the memory latency. In this context, the most important issues in packet processing are the answers to

'how the packets come in and get out of a chip and memory'; computation becomes a side issue.

The current technologies provide scalability in traffic processing by means of tightly-coupled parallelism (multi-core). Although there are powerful packet processing systems (e.g., network processors) that perform traffic processing applications at high-speed links in use now, we believe that application demands increase beyond the processing ability of only one stand-alone, single processing node system. Moreover, new hardware components best specialised for parts of the processed traffic (FPGAs) become available. The thesis of this dissertation is that processing traffic at future link rates is facilitated by parallel processing tasks either on single node, or for even higher rates, using multiple nodes in a heterogeneous distributed architecture.

Using distributed system addresses, at a minimum, two current demands. The first demand is *scalability*. More work needs to be done in the same time by only one node (e.g., traffic processing at multi-gigabit rates needed at the network edges in large community areas). The second demand is the use of *heterogeneous* systems shown by the possibility to use specialised systems for various and different traffic functions (e.g., IDS, generic traffic monitoring, video streams processing), or to use different hardware generations for the purpose of traffic processing.

One can say that using a parallel machine (e.g., many Pentium cores on the same die) can give the same processing power as a distributed system could do, but at a smaller development price (e.g., using better development environments and known trade-offs from the generic symmetric multi-processing research domain). However, getting good performance from parallel machines requires careful attention to data partitioning, data locality, and processor affinity. When using distributed systems, we care less about such low-level application partitions, and we focus on higher level issues like how to map the application on multiple nodes so that it processes the traffic optimally. For instance, in a processing hierarchy, the first nodes may offload the processing tasks of the last (leaf) nodes by pre-processing part of the entire task. For instance, a distributed intrusion detection system can use heterogeneous processing nodes such as old generation network processors for traffic splitting, and several state-of-the-art network processors for deep traffic inspection.

In addition to a distributed traffic system we show that we need systems that are able to manage themselves because the complexity of the systems (and applications) grows in time and the environment becomes more and more hostile at the same time. For example, when the traffic increases beyond the processing capacity of one node from the processing hierarchy then a supervisor may decide to offload the affected node by moving the task onto another node (having more resources) or by replicating the troubled task over multiple nodes.

1.5 Thesis overview

The thesis is outlined as follows:

Chapter 2 (*Background*) gives a brief presentation of the FFPF software framework used (and extended) for providing proof of concepts during this research. The text in this section is based on the FFPF paper of which the author of this thesis was a co-author. The chapter then describes the state-of-the-art hardware, network processors, that are largely used in networking applications at high-speeds at the present time.

Chapter 3 (*FPL: a Resource Constrained Language for Packet Processing*) introduces the FPL language and the FPL compiler as a support added to the FFPF software framework. The FPL language is a new packet processing language proposed as a means of obtaining the levels of flexibility, expressive power, and maintainability that such a complex packet processing system requires.

Parts of this chapter have been published in the *Proceedings of the 6th USENIX Symposium on Operating Systems Design* (OSDI'04).

Chapter 4 (*FPL Run-time Environments*) presents the run-time extensions made to the FFPF packet processing framework in order to support the FPL applications onto several hardware architectures in use in these days: commodity PCs, network processors, and FPGAs. Although the run-time environments make use of tightly coupled parallelism through multi-cores, they are limited to a single-node architecture.

Parts of this chapter have been published in the *Proceedings of the 5th Conference of the Embedded Computer Systems: Architectures, MOdeling, and Simulation* (SAMOS'05) and in the *IEEE Proceedings on IP Operations & Management* (IPOM'04).

Chapter 5 (*Distributed packet processing in multi-node: NET-FFPF*) introduces the distributed traffic processing concepts that form some of the key components in the extended multi-node FFPF framework. This chapter presents the distributed network processing environment and the extensions made to the FPL programming language, which enable users to process network traffic at high speeds by distributing tasks over a network of commodity and/or special purpose devices such as PCs and network processors. A task is distributed by constructing a processing tree that executes simple tasks such as splitting traffic near the root of the tree while executing more demanding tasks at the lesser-travelled leaves. Explicit language support in FPL enables us to efficiently map a program to such a tree.

Parts of this chapter have been published in the *Proceedings of the 4th International Conferences on Networking* (Networking'05).

Chapter 6 (*Control for Distributed Traffic Processing: ConDucTor*) introduces a control architecture for distributed traffic processing systems. The control architecture proposes a control loop that monitors and adjusts each processing node of the entire distributed system. In order to achieve the stability goal, the controller re-maps the application tasks from a congested node to another and re-distributes the traffic across the distributed processing hierarchy accordingly.

Chapter 7 (*A Simple Implementation of Conductor Based on Centralised Control*) presents an implementation of the centralised control approach applied to our distributed traffic processing system.

Chapter 8 (*Case study: Token Based Switch*) presents a case study in which the FPL compiler and the extended run-time version of the FFPF framework described in Chapter 3 are applied to build a specific application: traffic authentication at multi-gigabit speeds using hardware encryption support.

Parts of this chapter have been published in the *Proceedings of the 6th International Conferences on Networking* (Networking'07).

Chapter 9 (*Conclusions*) closes the thesis with a summary and discussion of the presented research topics, and concludes with some suggestions for future research.

Chapter 2

Background

In this chapter, we introduce the software framework – Fairly Fast Packet Filter (FFPF) – and the hardware – IXP network processors – used as a background support for the development of the tools needed to prove the concepts of parallel and distributed traffic processing.

2.1 The FFPF framework

Most network monitoring tools in use today were designed for low-speed networks under the assumption that computing speed compares favourably to network speed. In such environments, the costs of copying packets to user space prior to processing them are acceptable. In today's networks, this assumption is no longer true. The number of cycles available to process a packet before the next one arrives (the cycle budget) is minimal. The situation is even worse if multiple monitoring applications are active simultaneously, which is increasingly common as monitors are used for traffic engineering, intrusion detection, steering schedulers in GRID computing, etc.

In this section, we discuss the implementation of the fairly fast packet filter (FFPF) framework. FFPF introduces a novel packet processing architecture that provides a solution for filtering and classification at high speeds. FFPF has three ambitious goals: speed (high rates), scalability (in number of applications) and flexibility. Speed and scalability are achieved by performing complex processing either in the kernel or on a network processor, and by minimising copying and context switches. Flexibility is considered equally important, and for this reason, FFPF is explicitly extensible with native code and allows complex behaviour to be constructed from simple components in various ways.

On the one hand, FFPF is designed as an alternative to kernel packet filters such as CSPF [43], BPF [18], mmdump [44], and xPF [45]. All of these approaches rely on copying many packets to userspace for complex processing (such as scanning the packets for intrusion attempts). In contrast, FFPF permits processing at lower levels and may require as few as zero copies (depending on the configuration) while minimising context switches. On the other hand, the FFPF framework allows one to add support for any of the above approaches.

FFPF is not meant to compete with monitoring suites like Coralreef that operate at a

higher level and provide libraries, applications and drivers to analyse data [13]. Also, unlike MPF [19], Pathfinder [20], DPF [21] and BPF+ [22], the FFPF goal is not to optimise filter expressions. Indeed, the FFPF framework itself is language *neutral* and currently supports five different filter languages. One of these languages is BPF, and an implementation of `libpcap` [28] exists, which ensures not only that FFPF is backward compatible with many popular tools (e.g., `tcpdump`, `ntop`, `snort` [11,46]), but also that these tools get a significant performance boost (see the FFPF evaluation on Section 4.4). Better still, FFPF allows users to mix and match packet functions written in different languages.

To take full advantage of all features offered by FFPF, we implemented two languages from scratch: FPL-1 (FFPF Packet Language 1) and its successor, FPL. The main difference between the two is that FPL-1 runs in an interpreter, while FPL code is compiled to fully optimised native code. The FPL-1 language is briefly illustrated in Section 2.1.2 and the FPL language is described in Section 3.1 as a part of the thesis work.

The aim of FFPF is to provide a complete, fast, and safe packet handling architecture that caters to all monitoring applications in existence today and provides extensibility for future applications.

Some contributions of the FFPF framework are summarised below.

- We generalise the concept of a 'flow' to a stream of packets that matches arbitrary user criteria;
- Context switching and packet copying are reduced (up to 'zero copy').
- We introduce the concept of a 'flow group', a group of applications that *share* a common packet buffer;
- Complex processing is possible in the kernel or NIC (reducing the number of packets that must be sent up to userspace), while Unix-style filter 'pipes' allow for building complex flow graphs;
- Persistent storage for flow-specific state (e.g., counters) is added, allowing filters to generate statistics, handle flows with dynamic ports, etc.

2.1.1 The FFPF architecture

The FFPF framework can be used in userspace, the kernel, an IXP2xxx network processor, a custom FPGA hardware, or a combination of the above. As network processors and FPGAs are not yet widely used, and (pure) userspace FFPF does not offer many speed advantages, the kernel version is currently the most popular. For this reason, we use FFPF-kernel to explain the architecture here, and describe the network processor and FPGAs versions later in Sections 4.2 and 4.3, respectively.

To introduce the architecture and the terminology, Figure 2.1 shows an example scenario in which two applications (A and B) monitor the network. We assume, for illustration purposes, that they are interested in overlapping 'flows', with the definition of 'flow' as follows. A flow is a kind of generalised socket which is created and closed by an application in order to receive/send a stream of packets, where the packets match a set of arbitrary criteria. Examples include: 'all packets in a TCP connection', 'all UDP packets', 'all UDP packets

containing a worm signature plus all TCP SYN packets', etc. A flow is captured by a set of interconnected filters, where a filter is defined as a processing engine that at the very least takes a stream of packets as input and generates a new (possibly empty) stream of packets as output (in addition, it may produce statistics and perform sophisticated tasks). Connected filters form 'processing graphs' through which packets flow.

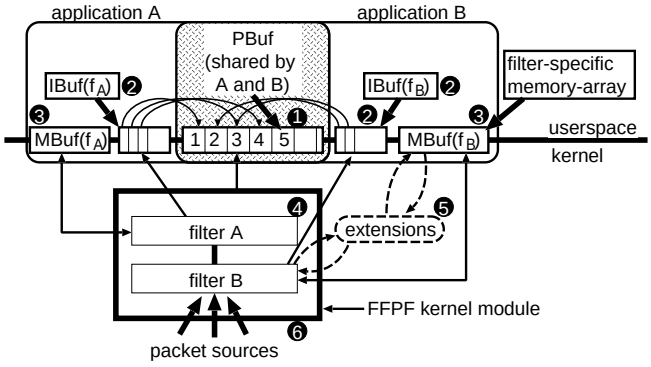


Figure 2.1: The FFPF architecture

The idea behind FFPF is simple (see Figure 2.1). In the kernel users load sets of connected filters that process packets ④. Strictly speaking, a *filter* is an instantiation of a *filter class*, and may perform more complex tasks than just filtering (e.g., it may generate statistics). The precise nature of the filters will be discussed in detail in Section 2.1.1. If a packet is classified by filter A as 'interesting', and it is not yet in a shared circular buffer (*PBuf*), it is pushed in *PBuf*, while a pointer to the packet is placed in filter A's index buffer ($IBuf_A$). If B is also interested in the packet, the packet is not copied, but rather another pointer to the packet is placed in B's index buffer ($IBuf_B$). Applications use the index buffers of filters to find the packets in which they are interested in *PBuf*. Clearly, proper management of the buffers is needed, but we skip the details in this section and return to this issue in Section 4.1.1. The third buffer, *MBuf*, is used for exchanging information between the application and the filter code in kernel space or to store persistent state. For instance, the expression may use it to store statistics.

All buffers are memory mapped, so in addition to avoiding copies to multiple applications, we also avoid copies between kernel and userspace. Depending on the language that is used for the filters, a filter expression may call extensions in the form of 'external functions' loaded either by the system administrator or the users themselves ⑤. External functions contain highly optimised native implementations of operations that are too expensive to execute in a 'safe' language (e.g., pattern matching, generating MD5 message digest).

Packets enter FFPF via one of the *packet sources* ⑥. Currently, three sources are defined. One is attached to the Linux netfilter framework. The second grabs packets even *before* they reach netfilter at the kernel's `netif_rx()` function. The third packet source captures packets from a network processor [47,48] and will be described in more detail in Section 4.2. Due to FFPF's modularity, adding more packet sources is trivial. In addition, depending on

the flow expressions, IPv4 and IPv6 sources can be mixed.

We will now summarise the relevant aspects of the FFPF architecture.

Flows

As said before, a key concept in FFPF is the notion of a *flow*. Flows are simply defined as a subset off all network packets. This definition is broader than the traditional notion of a flow (e.g., a 'TCP connection') and encompasses for instance all TCP SYN packets or all packets destined for the user with UID 0. To accommodate for such diverse flows, FFPF, instead of specifying filters itself, allows for varied selection criteria through extensions. This makes it more versatile than traditional flow accounting frameworks (e.g., NetFlow or IPFIX [49]). Furthermore, FFPF filters can be interconnected into a graph structure similar to that of the Click [50] router for even more fine-grained control. A filter embedded in such a graph is called a *flowgrabber*.

Grouping

The *flowgroup* constitutes a second key concept in FFPF. Flowgroups allow multiple flows to share their resources. As resource sharing poses a security hazard, group membership is decided by an application's access constraints. Network packets can be shared safely between all applications in a single flowgroup. Whenever a packet is accepted by one or more filters in a flowgroup, it is placed in a circular packet buffer (*PBuf*) only once, and a reference to this packet is placed in the individual filters' index buffers (*IBuf*). In other words, there is no separate packet copy per application. As buffers are memory mapped, there is no copy from kernel to userspace either. For instance, if a flow A is in the same flow group as another flow B it is allowed to read all packets captured by B. As a result, the packets can be read from the same buffer and need not be copied to each application individually.

Filter expressions

FFPF is language neutral, which means that different languages may be mixed. As mentioned earlier, we currently support five languages: BPF, FPL-1, FPL, C, and OKE-Cyclone. Support for C is limited to root users. The nature of the other languages will be discussed in more detail in Section 4.1. Presently, we only sketch how multiple languages are supported by the framework.

Figure (2.2.a) shows an example with two simplified flow definitions, for flows A and B, respectively. The grabber for flow A scans web traffic for the occurrence of a worm signature (e.g., CodeRed) and saves the IP source and destination addresses of all infected packets. In case the signature was not encountered before, the packet is also handed to the application. Flow grabber B counts the number of fragments in web traffic. The first fragment of each fragmented packet is passed to the application.

There are a few things that we should notice. First, one of these applications is fairly complex, performing a full payload scan, while the other shows how the state is kept regardless of whether a packet itself is sent to userspace. It is difficult to receive these flows efficiently using existing packet filtering frameworks, because they either do not allow complex processing in the kernel, or do not keep persistent state, or both. Second, both flows may

end up grabbing the same packets. Third, the processing in both flows is partly overlapping: they both work on HTTP packets, which means that they first check whether the packets are TCP/IP with destination port 80 (first block in Figure 2.2.a). Fourth, as fragmentation is rare and few packets contain the CodeRed worm, in the common case there is no need for the monitoring application to get involved at all.

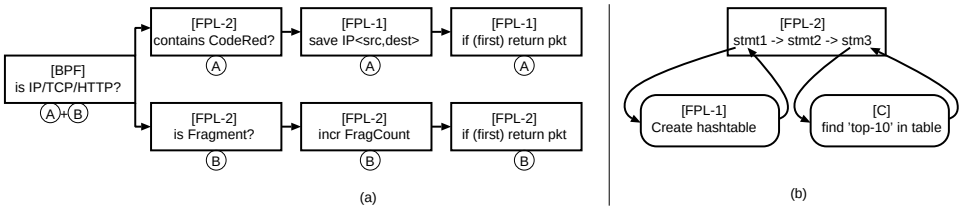


Figure 2.2: (a) combining different languages in two flows (A and B), (b) calling external functions from a single flow

Figure (2.2.a) shows how these two flows can be accommodated. A common BPF filter selecting HTTP/TCP/IP packets is shared by both flows. They are connected to the flow-specific parts of the data paths. As shown in the figure, the data paths are made up of small components written in different languages. The constituent filters are connected in a fashion similar to UNIX pipes. Moreover, a pipe may be 'split' (i.e., sent to multiple other pipes, as shown in the figure) and multiple pipes may even be 'joined'. Again, in UNIX fashion, the framework allows applications to create complex filter structures using simple components. A difference with UNIX pipes, however, is the method of connection: FFPF automatically recognises overlapping requests and merges the respective filters, thereby also taking care of all component interconnects.

Each filter has its own *IBuf*, and *MBuf*, and, once connected to a packet source, may be used as a 'flow grabber' in its own right (just like a stage in a UNIX pipe is itself an application). Filters may read the *MBuf* of other filters in their flow group. In case the same *MBuf* needs to be written by multiple filters, the solution is to use function-like *filter calls* supported by FPL-1 and FPL, rather than pipe-like *filter concatenation* discussed so far. For filter call semantics, a filter is called *explicitly* as an external function by a statement in an FPL expression, rather than implicitly in a concatenated pipe. An explicit call will execute the target filter expression with the calling filter's *IBuf* and *MBuf*. An example is shown in Figure (2.2.b), where a first filter call creates a hash table with counters for each TCP flow, while a second filter call scans the hash table for the top-10 most active flows. Both access the same memory area.

Construction of filter graphs by users

FFPF comes with a few constructs to build complex graphs out of individual filters. While the constructs can be used by means of a library, they are also supported by a simple command-line tool called `ffpf-flow`. For example, pronouncing the construct '`->`' as 'connects to' and '`|`' as 'in parallel with', the command below captures two different flows:

```
./ffpf-flow  "(device,eth0) | (device,eth1) -> (sampler,2,4) -> \
```

```
    (FPL-2,"...") | (BPF,"...") -> (bytecount,,8)"
"(device, eth0) -> (sampler,2,4) -> (BPF,"...") -> (packetcount,,8)"
```

The top flow specification indicates that the grabber should capture packets from devices `eth0` and `eth1`, and pass them to a sampler that captures one in two packets and requires four bytes of *MBuf*. Next, sampled packets are sent both to an FPL filter and to a BPF filter. These filters execute user-specified filter expressions (indicated by '. . .'), and in this example require no *MBuf*. All packets that pass these filters are sent to a bytecount 'filter' which stores the byte count statistic in an *MBuf* in an eight byte counter. The counter can be read directly from userspace, while the packets themselves are not passed to the monitoring application. The second flow has a prefix of two 'filters' in common with the first expression (devices are treated as filters in FFPF), but now the packets are forwarded to a different BPF filter, and from there to a packet counter.

As a by-product, FFPF generates a graphical representation of the entire filter-graph. A graph for the two flows above is shown in Figure 2.3. For illustration purposes, the graph shows few details. We just show (a) the configuration of the filter graph as instantiated by the users (the ovals at the top of the figure), (b) the filter instantiations to which each of the component filters corresponds (circles), and (c) the filter classes upon which each of the instantiations is based (squares). Note that there is only one instantiation of the sampler, even though it is used in two different flows. On the other hand, there are two instantiations of the BPF filter class. The reason is that the filter expressions in the two flows are different.

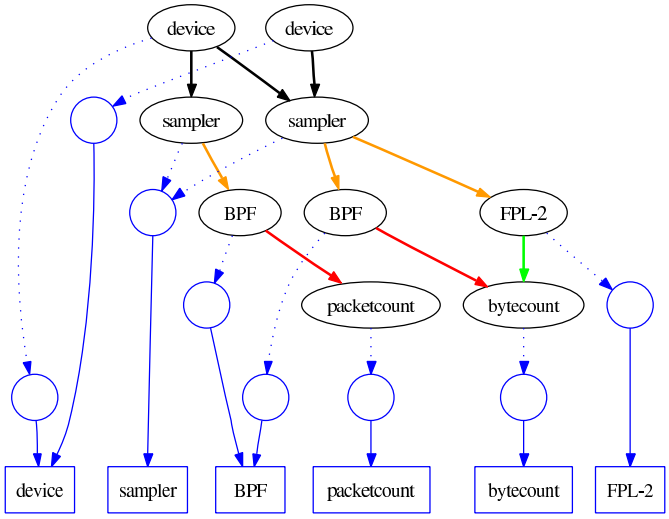


Figure 2.3: Auto-generated diagram of filter graph

The ability to load and interconnect high-speed packet handlers in the kernel was also explored by Wallach et al., with an eye on integrating layer processing and reducing copying [51]. Similarly, Click allows programmers to load packet processing functions consisting

of a configuration of simple elements that push (pull) data to (from) each other [50]. The same model was used in the Corral, but with support for third parties that may add and remove elements at runtime [52]. The filter concatenation and support for a hierarchy that includes IXP1200s resembles paths in the Scout project [30]. Scout was not designed for monitoring *per se* and, hence, does not directly provide some of FFPF's features such as new languages or flow groups. Mono-lingual kernel-programming projects that also do not support these features include FLAME [53] and Open Kernel Environment [54]) which provide high speed processing by loading native code (compiled Cyclone) in the kernel.

2.1.2 The FPL-1 language

As noticed before, the FFPF framework was designed to support multiple packet processing languages such as BPF [18] and FPL. FPL-1 is a new language that allows to write 'filtering' expressions which can be injected in the FFPF runtime. FPL-1 was one of the first attempt towards a packet processing language that combines the advantages of existing languages and also allows future extensions. The FPL-1 language elements are described below.

Filter expressions

Filtering means that any packet matching the filter will be added to the FFPF internal circular buffer. Expressions are postfix and relatively simple affairs, for example:

```
(subexpr1)(subexpr2)||(subexpr3)&&
```

This will evaluate to true as follows: subexpr3 is true and either subexpr1 or subexpr2 is true.

Operators

Operators allowed are:

1. '&&', '||'
2. '&', '|'
3. '<', '>', '<=', '>=', '=', '!='
4. '+', '-', '*', '/', '%'.

Operands

Besides the operators, FPL-1 has operands that are typed as follows: b=bit, c=octet, s=int16, i=int32, N='32bit const number'.

Notice that types b, c, s, i are followed by an offset to specify an instance in the data, e.g. 'c18' denotes the octet at index 18. If we need to indicate a constant number instead of some bit of data in the packet, use the 'N' type.

We note that brackets '()' and commas ',' have no real meaning whatsoever. We can use them, but they will be ignored in the evaluation process. Their only real use is to distinguish, for instance, a logical AND operation from a binary AND operation following it as in the following example:

```
# the following is ambiguous:
```

```
  ...,s10,N0,&&&... # & followed by &&, or && followed by &?
# so replace by
  ...,(s10,N0,&)&& # unambiguous: & followed by &&
```

However, the square brackets '[]' do have a special meaning, as illustrated later in the example section.

FPL-1 by examples

In order to show the way FPL-1 looks like, we present some examples which assume IP traffic on top of Ethernet, as follows:

1. a simple test for a UDP port: true if (int16 at offset 10) == (const number 54322)
   `(s10,N54322,=)`

2. a test for certain types of packets such as used in a drop filter: match every packet with 54321 in the 10th short (e.g. UDP port), 118 in the 28th byte, and a 'seqnum' that is not a multiple of 10
   `(s10,N54321,=)(c28,N118,=)(i8,N10,%)&&&&`

3. data as dynamic offset by using square brackets to define subexpressions. This allows us, for instance, to use a specific byte in the data as an offset. The following example matches all packets that have a value of 10 in the byte at the offset determined by the value of byte 1
   `(c[c1],N10,=)`

   Moreover, between the square brackets one can enter arbitrary new expressions, such as: `(c[(c1,N5,+),i[c3],*],N10,=)`

4. stateful memory: *MBuf*is implemented as an array of unsigned longs of specified length in which one may store temporary results. The format is: `(val,index,@)` to store 'val' in the array at position 'index' (val will be on top of the stack). Use `(index,$)` to push the value of the memory at position 'index' on the stack. In the following, the element at index i in the memory array will sometimes be written as M[i].

   - sample store: store 0 in the 7th element of the array (M[7])
     `(N0,N7,@)`
   - sample store: store 7 in the array element indexed by c10
     `(N7,c10,@)`
   - sample load: push M[0] on the stack (also returned)
     `(N0,$)`
   - sample load: return 1 if M[0] contains 7
     `(N0,$),N7,=`

   Note that in these memory operations, all parameters are always removed from the stack including, for instance, the value that you are writing into memory. The result is always pushed onto the stack. Example:

   - after the following expression the stack will contain 5: `N5,N1,@`

- after the following expression the stack will contain the value of array element 0: `N0,$`
- after the following expression the stack will be 5: `(N5,N1,@),N1,$`

5. hashing: the hash operation pops two values of the stack (offset and len) and calculates a hash over the data in the packet buffer starting at the 'offset' and for a length of 'len'. Both parameters should be given in bytes. For example, the following expression pushes a hash over the (ipsrc, ipdest) pair on the stack:
   `N11,N8,H`

6. for loops is a construct that allows one to implement simple loops (either ascending or descending). The loop variable is kept somewhere in the memory array, at an index determined by the user complying with a stack-based approach. The format is as follows:
   `(index,startval,endval,stepsize,FOR) ... FBR ... FEND`,
   where:

   - 'index' is the index in the memory which identifies the cell to use as loop;
   - 'startval' is the initial value of the loop;
   - 'endval' is the target value of the loop. Note that if endval $<$ startval then the loop is descendent;
   - 'stepsize' is the step used to increase or decrease the loop counter;
   - FOR: We notice that the for loops in FPL-1 cannot be nested;
   - FBR: break from for loop, save to stack, unwind stack, push saved value, and continue with the expression following FEND;
   - FEND: check if startval has overtaken endval; if so, do as in FBR; if not, continue with the expression following FOR.

   It is clear that for loops like these do not match a simple expression language well. However, it allows us to build very powerful expressions (e.g., one that searches for a substring). Here is a trivial example:

   Test memory and loop: if the first byte in the packet is a 1, count all zeros in the first 10 bytes. Maintain the loop variable in M[0] and the 'zero counter' in M[1]:

```
(c0,N1,=)(N0,N1,@)(N0,N0,N10,N1,FOR)
 (((c[N0,$],N0,=)(N1,$)+),N1,@)(FEND)+*"
```

7. external functions: It is possible to call external functions (previously registered) from FFPF filter expressions.
   `(X'HelloFn')`

   Note that external functions are potentially powerful and fast. For example, we may use an external function to check whether we have received a real-time signalling protocol (RTSP) packet and if so, we *change the filter* to find also all corresponding multimedia 'data' packets. These data packets tend to be UDP packets sent to a set of dynamic port numbers and which therefore, could not easily be captured by a static filter.

8. return from subexpression: It is possible to return from any subexpression to the expression one level up by way of the return instruction 'R'. Upon encountering the return instruction, the top of the stack is popped and returned as the result of the subexpression. Execution resumes at the statement following the end of the subexpression. This can be either a real end-of-subexpression marker (']'), or the end of the top-level expression if we are executing at the highest level already.

   For example, return after the first add, and do not execute any of the remaining instructions. The result will be 2.
   `(N1,N1,+),R,N5,*,3,2,+,-`

Summarising, FPL-1 is a powerful language by providing operators and operands to build expressions in order to handle any packet data. Moreover, it is also extensible by means of external functions concept. However, the language itself is not easy to use due to its stack based implementation and arcane syntax. In this respect, we will introduce the successor, the FPL language, in Section 3.1. FPL provides a more 'intuitive' way to write packet processing applications and better performance by using a new tool, the FPL-compiler, to compile the user expressions into 'native' code for a specific hardware architecture target.

2.2 Network Processors

A network processor (NP) is an Application Specific Instruction Processor (ASIP) for the network application domain. In other words, an NP is a programmable device with architectural features for packet processing. As a result, network processors (NPs) share characteristics with many different implementation choices:

- communication processors used for networking applications;
- 'programmable' state machines used for routing;
- reconfigurable fabrics (e.g., FPGAs);
- general purpose processor (GPP) used for configuration, routing.

Network processors appeared as a solution to the needs for customisability, in-the-field programmability, and shrinking time-to-market windows in the network processing application domain.

In the following sections we present shortly the common hardware characteristics of NPs, and then we describe the Intel network processors.

2.2.1 Common characteristics in NPs

In the last few years, many silicon manufacturers released NPs [47, 48, 55–62] with architectures designed to suit specific network configurations, types of packets being processed, speed of the network physical layer, multithreading capability, co-processor support, memory types with various access speeds, etc. Although the NPs are made of various components with different configurations depending on the design specifications, a common objective is

to process network packets at wire speed for the most used packet processing tasks, and give programmable flexibility to adopt fast evolving protocol definitions.

The common components of a NP are:

- programmable processing elements (PE) with or without multithreading capabilities;
- on-chip memories with different access speeds (e.g., local mem, scratch, registers);
- external memory interfaces with different access speeds and sizes such as SRAM and DRAM;
- function specific co-processors or ASICs such as hashing, or encryption units;
- external network connection bus and interfaces (e.g., SPI);
- Switch fabric support for interconnection to other packet processor units (NPs, FPGA processors, etc.);
- connection to an external general processor such as a PCI bus interface;
- hardware specific instruction set oriented for packet processing (no floating point units for instance);
- compilers/assemblers;
- debugging and programming tools with some GUI support.

For the purpose of this research we have chosen the Intel IXP network processors because we had access to the (expensive) hardware and they emerged as one of the most popular choices for the research world. In the next sections we describe the Intel network processors dedicated to high speed packet processing and used during this research. The first generation was opened by IXP1200, then the second generation includes IXP2400 and IXP28xx.

Comparing the evolution of Intel IXP NPs in a total MIPS benchmark chip, we note that the first generation, IXP1200 – 1999, provided 1200 MIPS, the second generation, IXP2400 – 2002, provided 4800 MIPS, and the last generation, IXP28xx – 2004, provides 23000 MIPS.

2.2.2 The IXP1200 processor

The Intel IXP1200 runs at a clockrate of 232 MHz and is mounted on a PCI board, in our case the Radisys ENP2506, together with 8 MB of SRAM and 256 MB of SDRAM. The board contains two Gigabit network ports. Packet reception and packet transmission over these ports is handled by the code on the IXP1200 processor. The Radisys board is connected to a Linux PC via a PCI bus. The IXP itself consists of a StrongARM host processor running embedded Linux and six independent RISC processors, known as microengines (see Figure 2.4). Each μEngine has its own instruction store and register sets. On each of the μEngines, registers are partitioned between four hardware contexts or 'threads' that have their own program counters and allow for zero-cycle context switches.

The IXP1200 consists of the following elements (see Figure 2.4) which reside on the same die:

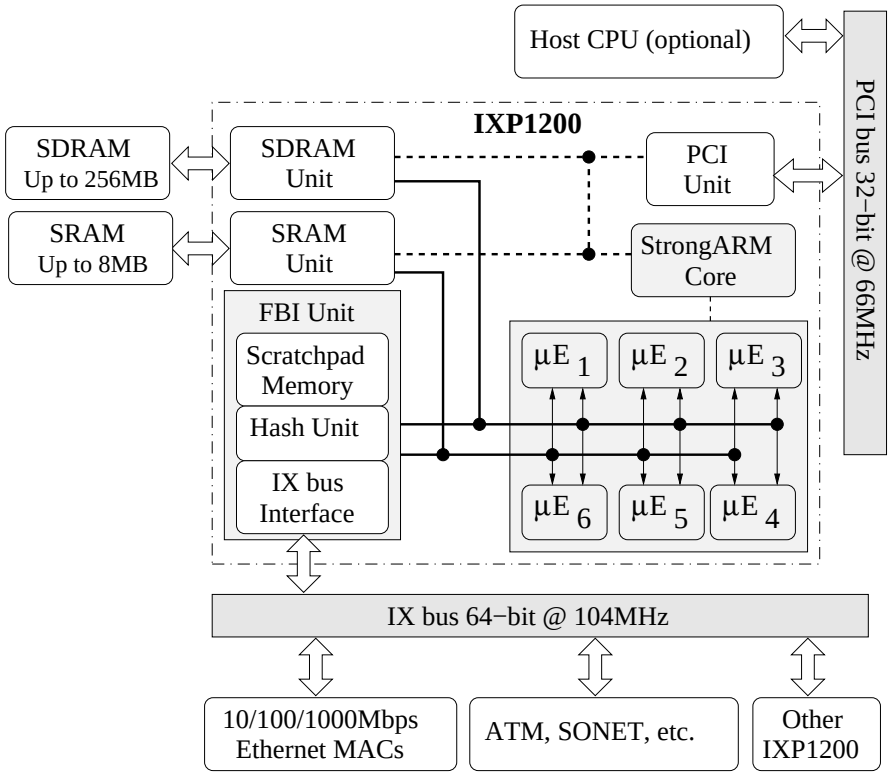


Figure 2.4: The IXP1200 NP block diagram.

- *StrongARM core*: It has 16 KBytes of instruction cache and 8 KBytes of data cache. The StrongARM is commonly used to initialize the chip, to handle exceptional packets, to communicate with other processors etc.;
- *Microengines*: The IXP1200 has six simple RISC processors used for packet processing at line rate. Each μEngine operates at the same frequency as the StrongARM and contains a large number of registers so that it supports four threads with zero-cycle context switches;
- *SDRAM Unit*: A unit for interfacing to external SDRAM modules;
- *SRAM Unit*: A unit for interfacing to external SRAM modules;
- *PCI Unit*: The unit for interfacing to PCI bus;
- *Hash Unit*: The unit implementing customizable hash functions;
- *Scratchpad*: 4 KBytes of internal SRAM, called 'Scratchpad';

- *IX Bus Unit*: A bus unit used for interfacing external devices such as fibre optic or copper PHYs, other NICs, etc. ;
- *FBI Unit*: The FIFO Bus Interface is a unit which hosts the IX bus interface, the Scratchpad memory and the Hash unit;
- *Interconnection Bus*: Separate buses for on-chip communication as illustrated in Figure 2.4.

Each μEngine supports up to four microcode context threads. Each μEngine thread has 32 general-purpose registers (GPRs) and a microprogram counter. The GPRs can be shared between threads of a μEngine or dedicated to a single thread. Thread context switches occur without delay, for instance, a μEngine can be executing one cycle in one thread and on the next cycle switch to another thread. The μEngines have an instruction store of 2K microwords each. μEngines can queue requests to memory on separate buses, enabling parallel access and maximum use of available memory bandwidth.

The IXP1200 network processor combines the best attributes of a network ASIC with the flexibility, performance, and scalability of a programmable embedded processor to accelerate development of the next-generation Internet products. The IXP1200 is considered to be the first Intel network processor generation. It was followed by the second generation: IXP2xxx.

2.2.3 The IXP2400 processor

Figure 2.5 shows the hardware architecture of the IXP2400 network processor and the functional blocks are described in the next paragraphs.

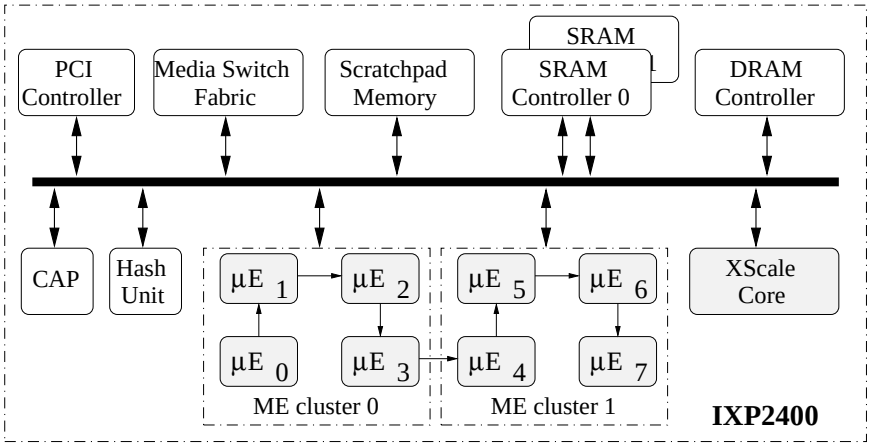


Figure 2.5: The IXP2400 NP block diagram.

Intel XScale core. The control processor is a general-purpose 32-bit RISC processor compatible to ARM Version 5 Architecture. Since the XScale core is a common ARM compliant

CPU, we run an ARM cross-compiled Linux for big endian. The XScale core is used to initialise and manage the chip, and can be used for higher layer network processing tasks.

Microengines (μEngines). The parallel architecture consists of 8 x 32-bit programmable engines (RISC processors) specialised for packet processing in parallel. μEngines do the main data plane processing per packet. A μEngine has hardware support for eight threads by providing all logic and registers for zero-overhead context switching. A control application running on the XScale core initialises and loads the μEngines' code. The μEngines in the IXP2400 have the following enhancements over the IXP1200 μEngines: (1) multiplier unit, (2) pseudo-random number generator, (3) CRC calculator, (4) 4x32-bit timers and timer signaling, (5) 16-entry CAM for inter-thread communication, (6) timestamping unit, (7) generalized thread signaling, (8) 640x32-bit words of local memory, (9) μEngines are divided into two clusters with independent command and SRAM buses.

Memory controllers. A network processor offers a good trade-off of cost/performance and hence, it combines different memory technologies such as expensive on-chip memory (e.g., registers, local memory), less expensive off-chip memory such as SRAM, and plenty of the cheap external DRAM memory. Combining these different memory types in a programmable manner so the user can choose where the program variables are located is successfully done with the help of the following controllers:

- *DRAM Controller*: one DDR SDRAM controller. Typically DRAM is used for data buffer storage;
- *SRAM Controller*: two independent controllers for QDR SRAM. Typically SRAM is used for control information storage;
- *Scratchpad Memory*: 16 KBytes of on-chip memory for general-purpose use.

Media and Switch Fabric Interface (MSF). MSF is an interface for network framers and/or Switch Fabric. Contains receive and transmit buffers.

Hash Unit. The IXP2400 has a polynomial hash accelerator which the XScale core and μEngines can use it to offload hash calculations.

PCI Controller. The PCI controller provides a 64-bit PCI bus which can be used to either connect to a host processor, or to attach PCI-compliant peripheral devices.

Control and status register Access Proxy (CAP). CAP contains the chip-wide control and status registers. These provide special inter-processor communication features to allow flexible and efficient inter-μEngine and μEngine-to-XScale-core communication.

The most visible high-level improvements of IXP2400 over its predecessor, IXP1200 are described in Table 2.1:

	IXP1200	IXP2400
control processor	StrongARM 233MHz	XScale 600MHz
μEngines	6 μEngines @ 233MHz	8 μEngines @ 600MHz
media switch fabric	FBI	MSF
receive/transmit buses	shared buses	separate buses

Table 2.1: IXP2400 versus IXP1200.

2.2.4 The IXP2850 processor

Figure 2.6 shows the hardware architecture of the last Intel network processor generation up to date: IXP2850.

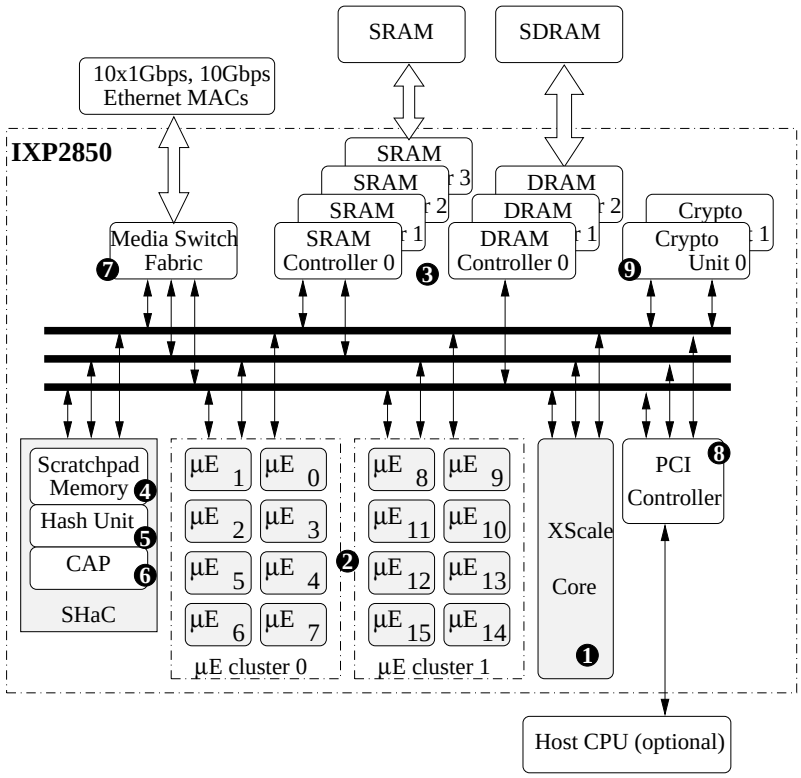


Figure 2.6: The IXP2850 NP block diagram.

The major IXP2850 blocks, also seen in the previous IXP architecture, IXP2400, are: ① XScale core, ② μEngines, ③ SRAM and DRAM Controllers, ④ Scratchpad and Local Memory, ⑤ Hash Unit, ⑥ Control and Status Register Access Proxy (CAP), ⑦ Media and Switch Fabric Interface (MSF), ⑧ PCI Controller.

The most notable advances in IXP2850 against the previous NP concerns the 10 Gbps processing capability and also the encryption/decryption hardware support ⑨. For instance,

IXP2850 has built-in two crypto units each containing two Triple Data Encryption Standard cores (3DES), two Secure Hash Algorithm cores (SHA-1), one Advance Encryption Standard core (AES) and one checksum unit.

The most visible high-level enhancements of IXP2850 over its predecessor, IXP2400 are described in Table 2.2.

	IXP2400	**IXP2850**
μEngines	8 μEngines @ 600MHz	16 μEngines @ 1.4GHz
SRAM controllers	2	4
DRAM controllers	1, DDR	3, Rambus
media switch fabric	SPI-3	SPI-4.2
cryptography engines	–	2x cryptography blocks that provides hardware acceleration of popular encryption and data integrity algorithms

Table 2.2: IXP2850 versus IXP2400.

Chapter 3

A Resource Constrained Language for Packet Processing: FPL

In this chapter we introduce a new language for packet processing in order to provide a programmable system capable of traffic monitoring at high speeds (Gbps). The programmability is offered with the help of a filtering language and a compiler for several hardware architectures in use currently: commodity PCs, network processors, and FPGAs. This chapter represents a first step towards handling high speed traffic by using tightly coupled parallelism through multi-cores.

The fairly fast packet filter (FFPF), as described in Section 2.1, is a packet processing architecture that provides a solution for filtering and classification at high speeds with three goals: speed (high rates), scalability (in number of applications) and flexibility. Speed and scalability are achieved by performing complex processing either in the kernel of a commodity PC or on specialised hardware such as a network processor, and by minimising copying and context switches. FFPF is explicitly extensible with native code and allows complex behaviour to be constructed from simple components.

3.1 The FFPF packet language: FPL

The need for a new packet processing language came together with the needs to provide a safe and easily programmable packet processing framework. Safety implies that the language must provide fairly strict control over resources, e.g., in order to prevent programs from consuming more cycles than available at link rate. It also implies that the user-provided code that may run in the lowest levels of the processing hierarchy is not able to access data or function for which it has no authorisation. Other requirements such as hiding to a user the hardware complexity, or the performance of the compiled code are equally important.

3.1.1 From interpreted to compiled code

FFPF was initially developed with a stack-based language called FFPF packet language 1: FPL-1. FPL-1 was difficult to program as it has an arcane syntax that explicitly expresses its stack-based implementation. Moreover, its mix of stack-based language and "conventional" for loops is somewhat counter-intuitive. Moreover, FPL-1 would not be efficiently supported by special purpose hardware such as network processors because the FPL-1 runtime uses an interpreter.

FPL-1 is a low-level stack language with support for most simple types and all common binary and logical operators, as briefly described in Section 2.1.2. In addition, FPL-1 provides restricted looping and explicit memory operations to access the persistent memory array. Flow expressions in FPL-1 are compiled to byte code and inserted in the FFPF kernel module by means of a special purpose code loader. Each time a packet arrives on one of the monitored interfaces, the FPL-1 expression of a flow f is called to see whether or not f is interested in this packet.

The disadvantage of FPL-1 is that it is slow as the filter's byte-code runs in an interpreter. It is difficult to compete both in writing efficient code optimisers and in providing an efficient packet processing environment, so rather than aiming for the best possible code generator we have chosen to exploit the optimisers of existing compilers for each supported hardware architecture. In other words, we use a source-to-source compiler followed by a back-end compiler for each specific hardware architecture. For instance, when using a commodity PC architecture, the FPL compiler generates C code that is subsequently compiled by `gcc` to a Linux kernel module that interfaces with the FFPF's run-time environment.

Summarising, we see an opportunity for a better language that addresses the following requirements: (1) safety against malicious or buggy code, (2) intuitive usage of language constructs, (3) hide specific hardware complexity, (4) extensibility, and (5) support for multiple hardware architectures. Although the next packet language generation would normally be called FPL-2, for the sake of naming simplicity we call the second packet language generation just FPL throughout the remainder of this thesis.

3.1.2 The FPL language

In the following subsections, we describe the support that was added for FPL, a new language that (1) compiles to fully optimised object code, and (2) is based on registers and memory.

Operators, expressions and statements in FPL

Operators act upon operands and specify what is to be done to them. Expressions combine variables and constants to create new values. Statements are expressions, assignments, external function calls, or control flow statements which make up filter programs. As illustrated in Table 3.1, most of the operators/expressions are designed in a way that resembles well-known imperative languages such as C or Pascal. Therefore, the users should easily adopt this filter language.

operator-type	operator
Arithmetic	`+, -, /, *, %, --, ++`
Assignment	`=, *=, /=, %=, +=, -=` `<<=, >>=, &=, ^=, |=`
Logical/Relational	`==, !=, >, <, >=, <=,` `&&, ||, !`
Bitwise	`&, |, ^, <<, >>`

statement-type	operator
if/then	IF (expression) THEN statement FI
if/then/else	IF (expression) THEN statement1 ELSE statement2 FI
for()	FOR (initialise; test; update) statements; BREAK; statements; ROF
external function	INT EXTERN(STR function_name, INT sharedMem_IndexRead, INT sharedMem_IndexWrite)
return a value	INT RETURN (val)
hash()	uint64 HASH(INT start_byte, INT len, INT range)

Table 3.1: Operators, expressions, and statements in FPL.

Memory data addressing

Memory addressing is a language element used for accessing the memory locations. We support two types of memory: a shared memory array `'M[]'` and fast local registers `'R[]'`.

The shared memory, declared in a filter program by `'MEM[size]'` statement, is provided by the FFPF core at filter initialisation by mean of an *MBuf*. Therefore, the filter module does not perform dynamic memory allocation/deallocation. *MBuf* consists of a byte array of a specified size. The shared memory can be used for various purposes. First, a shared memory is used for sharing the processing results in a filter to a higher level application (e.g., a user interface). Second, in a stateful packet processing demand, a shared memory can be used for storage between consecutive calls of the filter on different packets. Third, we will see how shared *MBufs* can be used for data exchange between filters.

The second supported memory type, registers allocated by the `'REG[size]'` statement, are general purpose registers available on the target hardware (e.g., x86 CPU, network processor). Generally, using data stored in registers increases the processing speed in case of very often used variables. The maximum number of local registers is defined in a resource configuration and depends on the hardware. However, we emphasise that placing something in registers serves as a hint only in some architectures.

As shown below, a memory location is easily accessed (retrieved from/stored in) by using the assignment operator.

```
1  MEM[1024];              // ask for a shared memory of 1024 of INT
2  REG[2];                 // ask for two local registers
3  M[9]=123+45*6/7-M[8];   // memory access
4  R[0]=M[8]%R[1];         // register access
5  R[1]=10+R[1];           // compute a relative index
6  M[R[1]] += 10;          // access a shared memory location at a variable
       index computed beforehand
```

Packet data addressing

Another language element is the access to the packet stream. Specific to our system is that regardless of how much parallelism the hardware system supports, at any time, a filter instance processes one packet: the current packet. This current packet is provided by the FFPF framework through the `'PKT.type[offset]'` statement by means of an offset to a start pointer of the packet. By choosing the type of the offset we can extract any bit of data within the packet data, as shown in Table 3.2. For improving the readability of programs, we use the lexical conventions according to the industrial standard IEC 1131-3 [63]. For instance, a byte (8 bits), word (16 bits), or a dword (32 bits) out of the packet at a specified offset are expressed by `PKT.B[]`, `PKT.W[]`, and `PKT.DW[]`, respectively. Moreover, we can retrieve a subset of the main offset by means of a sub-index specified through `.U1`, `.U4`, `.U8`, `.U16`, for bit, nibble, byte, or word, respectively.

Data type	**syntax**
Memory access:	
Register n	R[n]
Shared memory location n	M[n]
Packet access:	
-byte $f(n)$	PKT.B[$f(n)$]
-word $f(n)$	PKT.W[$f(n)$]
-double word $f(n)$	PKT.DW[$f(n)$]
-bit m in byte n	PKT.B[n].U1[m]
-nibble m in byte n	PKT.B[n].U4[m]
-bit m in word n	PKT.W[n].U1[m]
-byte m in word n	PKT.W[n].U8[m]
-bit m in dword n	PKT.DW[n].U1[m]
-byte m in dword n	PKT.DW[n].U8[m]
-word m in dword n	PKT.DW[n].U16[m]
-macro	PKT.macro_name
-ip proto	PKT.IP_PROTO
-ip length	PKT.IP_TOTAL_LEN
-etc.	customised macros

Table 3.2: Memory & packet data addressing modes.

In addition to direct access to the packet fields by using offsets, as described before, there is also a possibility to use macros for the most used packet fields. A macro is previously defined in a compiler configuration file and then is used in any filter program by

'PKT.macro_name' statement. Some examples of macros that work on a system where FFPF provides the start packet pointer at the IP layer are drawn in Table 3.3. These examples show how easy and intuitive a specific IP field is reached within the packet. Summarising, macros allow users to customise the way they express themselves and also to hide non-uniform implementation aspects between different hardware systems. For instance, the start packet pointer in a commodity PC points to the IP layer while in a network processor it points to the Ethernet layer.

Moreover, using a register or memory variable as index for packet reference the language increases considerably the applications area. In the (trivial) example shown below, the sum of the first 20 bytes in the packet is computed.

```
1  FOR (R[0]=0;R[0]<20;R[0]++)
2    M[0]+= PKT.B[R[0]];
3  ROF
```

Macro name	operator in IP layer	result
IP_VERS	PKT.B[0].HI	U8 (the value of high four bits of the first byte)
IP_HDRLEN	PKT.B[0].LO	U8 (the value of low four bits of the first byte)
IP_TOS	PKT.B[1]	U8 (the value of whole byte)
IP_TOTAL_LEN	PKT.W[1]	U16 (the value of second word)
IP_DATAGRAM	PKT.W[2]	U16 (the value of third word)
IP_FRAGM	PKT.W[3]	U16 (the value of fourth word)
IP_TTL	PKT.B[8] or PKT.W[4].HI	U8
IP_PROTO	PKT.B[9] or PKT.W[4].LO	U8
IP_HDRCHK	PKT.W[5]	U16
IP_SRC	PKT.DW[3]	U32
IP_DEST	PKT.DW[4]	U32
UDP_SRC	PKT.DW[5].HI or PKT.W[10]	U16
UDP_DEST	PKT.DW[5].LO or PKT.W[11]	U16
UDP_LEN	PKT.DW[6].HI or PKT.W[12]	U16
UDP_CHECKSUM	PKT.DW[6].LO or PKT.W[13]	U16

Table 3.3: Examples of macros for IP packet fields.

HASH() construct

Hash functions are an efficient way to compute a 'unique' identifier on a given input data set (e.g., a string or a byte array). In software development, hashing is mostly used to build so-called hash tables because they offer an excellent average search time. For instance, Linux relies on hash tables to manage pages, buffers, and other data objects. In networking, we

use hash tables to manage routing tables. The construct is often used for getting a 'unique' identifier of packets that belong to the same network user (e.g., by hashing the src/dest IP addresses of the packets) or that belong to the same application (e.g., by hashing also src/dest port addresses in addition to the IP addresses).

The hash construct in our FPL language applies a hash function to a sequence of bytes in the packet data provided as input parameters. However, when using different architectures we might have different hash implementation for this construct. For example, in a commodity PC running Linux, the hash function used is a simple software implementation using Horner's rule to avoid expensive computation such as handling coefficients with high values in power. Besides software implementations, we have also hardware-accelerated hash functions available in special purpose processors such as network processors. Such hardware-accelerated hash functions use a hard-wired polynomial algorithm and a programmable hash multiplier to create hash indices. For instance, the Intel IXP2400 network processor supports three separate multipliers, one for 48 bit hash operations, one for 64 bit hash operations and one for 128 bit hash operations. The multiplier is programmed through control registers. For the sake of compatibility and simplicity, our FPL hash construct uses a 64 bit implementation on commodity PCs and network processors and hence, we expect a 64 bit hash result.

An example of using the hash construct is given below. Assuming we have a hash implementation on network processors, we need a 'unique' identifier in order to count TCP flows. This construct applies over a byte array starting at the specified offset in an Ethernet frame (byte 26th) until a specified length long enough to cover IP source/dest addresses and TCP source ports (12 bytes).

```
1  R[0] = Hash(26, 12, 255); // hash over TCP flow fields and store into a
       register the truncated result for the specified range [0-255]
```

In practice, a hash result is often used as an index into a table (e.g., a routing or a status table). As such tables are limited in size, an optional range parameter can be used to truncate the value efficiently. Although our hash uses a 64 bit implementation, other hashes can be added by way of external functions, as it will be shown later.

External functions

An important feature of FPL is extensibility and the concept of an 'external function' is key to extensibility, flexibility and speed. External functions are an explicit mechanism to introduce extended functionality to FFPF and also add to flexibility by implementing 'call' semantics. While they look like filters, the functions may implement anything that is considered useful (e.g., checksum calculation, pattern matching). They can be written in any of the supported languages, but it is anticipated that they will often be used to call optimised native code performing computationally expensive operations.

In FPL, an external function is called using the `EXTERN` construct, where the parameters indicate the filter to call, the offset in a shared buffer *MBuf* where the filter can find its input data (if any), and the offset at which it should write its output, respectively. For instance, `EXTERN(foo,x,y)` will call external function `foo`, which will read its input from memory at offset x, and produce output, if any, at offset y. Note that FFPF does not prevent users from supplying bogus arguments, nor using buggy external functions when written in other language than FPL. However, for FPL language written filters, the protection comes from

the FPL-compiler. The compiler checks the use of external functions in a filter. An external function's definition prescribes the size of the parameters, so whenever a user's filter tries to let the external function read its input from an offset that would make it stray beyond the bounds of the *MBuf* memory array, an error is generated. This is one of the advantages of having a 'trusted' FPL-compiler (see also Section 3.1.4). In addition, authorisation control can be used to grant users access only to a set of registered functions.

A small library of external filter functions has been implemented (including implementations of popular pattern matching algorithms, such as Aho-Corasick and Boyer-Moore). The implementation will be evaluated in Section 4.4.1.

Shared memory usage

In the FFPF architecture, there are many interfaces related to shared memory, but only two of them are relevant for the packet filtering language and thus they are described presently.

The first interface involves the data exchange between the FFPF core and each loaded filter module as illustrated in Figure 3.1. As an example, the results of a particular filter can be periodically read by a user application.

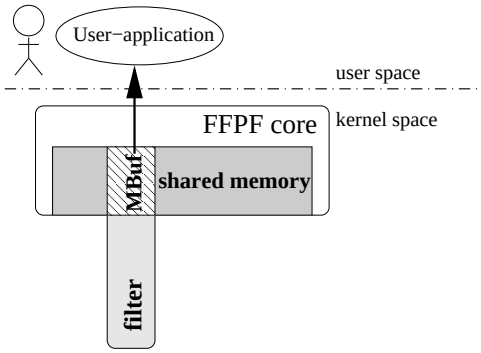


Figure 3.1: Shared memory between user applications and FPL filters.

The second interface is used for data exchange between filters with the help of 'external' functions. Assume that there are two filters `'Foo'` and `'Bar'` already registered within the FFPF core, and one needs to access the processing results of the other. As shown in Figure 3.2, the filter 'Foo' accesses part of the shared memory of the filter 'Bar' when it issues an external call like:

```
EXTERN(bar, readIndex, writeIndex).
```

Another example of *MBuf* usage as persistent state in FPL is shown in Listing 3.1. The code describes a filter that keeps track of how many packets were received on each TCP connection (assuming for simplicity that the hash is unique for each live TCP flow). The *MBuf* is shared with the user application for efficient access to end results.

Restricted FOR loops

The `FOR` loop is a construct used for running repetitively a piece of code such as searching for a pattern in a certain range of bytes of the current packet. Although `FOR` loops are useful

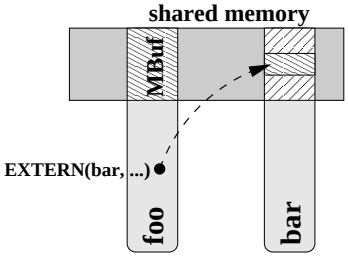


Figure 3.2: Shared memory between filters through external functions.

```
1  // count number of packets in every flow by keeping counters in a hash
       table (assume hash is unique for each flow)
2  MEM[4095];
3  IF (PKT.IP_PROTO == PROTO_TCP) THEN
4    R[0] = Hash(26, 12, 4095);  // register = hash over TCP flow fields
5    MEM[ R[0] ]++;              // increment the pkt counter at this position
6  FI;
```

Listing 3.1: Example of FPL code: count TCP flow activity.

in a program, they are also dangerous in our specific domain, packet processing, because of the limited cycle budget available for the execution of the entire program. The cycle budget is determined by the wire speed and hence, by the inter-arrival packet time. Therefore, the higher the wire-speed is, the smaller the cycle budget is.

Some existing packet languages such as SNAP [64] do not permit ANY backward jumps in the program. In contrast, we allow loops but restrict the number of iterations. FOR loops can easily be used to write malicious code or belong to buggy code. Because our resources are strictly limited in terms of processing time, in order to avoid packet loss, the `FOR` loop construct is limited to loops with a pre-determined number of iterations.

Users specify both start and end values of the iteration variable, as well as the amount by which the loop variable should be incremented or decremented after each iteration. The `BREAK` instruction, allows one to exit the loop 'early'. In this case (and also when the loop finishes), execution continues at the instruction following the `ROF` construct. `FOR` loops can be used to test a small range of bytes in the packet or even to scan the entire packet payload for the occurrence of a pattern. Although we allow execution of external functions within FOR loops, we assume that the functions behave correctly regardless of the language was written. In other words, the FPL-compiler cannot guarantee the 'correctness' of functions written in different languages. Recall, however, that we are able to control which EXTERNAL functions a user can call.

However, the FOR loops are 'restricted' in the sense that they are bounds-checked at runtime against a maximum number of allowed iterations (e.g., 1000) provided in a compiler definition or using a credential scheme as described in Section 3.1.5. In essence, there is an upper bound on the total number of iterations of all loops in the program. This bound-check applies also to nested FOR loops, as shown in the example in Listing 3.2. We take a conservative approach and assume loops run for the maximum number of iterations. Although

breaks within loops are allowed and hence a loop may end prematurely, the for-loop bound-check always takes into account the entire amount of iterations which a loop must execute. One could say that the compiler can bound-check the FOR loops easily, at compile time. However, we notice that FOR loops need to be bound-checked at runtime because processing packets often involves searching for a pattern within the packet payload. In other words, some loops may run for a variable maximum iterations such as the size of a worm signature, or the total packet length. In our example shown in Listing 3.2 we show a simple loop which runs until PKT.IP_TOTAL_LEN and for illustration purpose we assume that the total IP length is 256 Bytes.

```
1  REG[2];
2  MEM[8];
3  R[1] = PKT.IP_TOTAL_LEN // e.g., IP_TOTAL_LEN is 256
4  FOR (R[0]=0; R[0]<R[1]; R[0]++)
5    M[0]++;
6    FOR (R[2]=8; R[2]>0; R[2]--)
7      M[1]++;
8      R[3]=1;
9    ROF;
10 ROF;
11 M[2]++;
12 FOR (R[3]=0; R[3]<10; R[3]++)
13   M[2]++;
14   R[5]=1;
15 ROF;
```

Listing 3.2: FOR loops in FPL.

The bound checks in the FOR loop constructs of the previous FPL example code are illustrated in the 'translated' code in Listing 3.4 for a commodity PC architecture (Linux OS). The checks consists of the helpers (`checkLoops()` function calls) that are introduced by the compiler automatically when it detects FOR constructs. The compiler generates a specific `CheckLoops()` function that provides, at any moment, the total number of executed iterations. The `CheckLoops()` function is specific to every FPL program, depending on the loops used by the FPL program (nested, simple, etc.). The FPL-compiler makes use of the Erhart's polynomials theory in order to implement a generic formula to compute the total amount of executed loops [65]. For instance, Listing 3.3 shows the implementation of `CheckLoops()` function for the example presented in Listing 3.2.

```
for (i=0; i<256; i++)
{
  for (j=8; j>0; j--)
  {
    foo(); //...
  }
}
// ...
for (k = 0 ; k <10 k++)
{
  bar(); // ...
}
/************ Check Loops formula *************************/
CheckLoops = ((i*8) + (8-j+1)) + (k+1);
```

Listing 3.3: Checking loops mechanism.

In the tailored C code example illustrated in Listing 3.4, the registers are used through the `g_Registers[]` variables, and the shared memory is pointed to by `g_MemPublic.lBuffer[]` variables. We notice that the bound checking variable for the for-loops is automatically generated by the FPL compiler and is hidden to the user. Therefore, the users cannot simply run infinite loops by changing the loop variable that they see.

```
1  int CheckLoops(void) {return g_Registers[0] * 8 + 8 - g_Registers[2]+ 1 +
       g_Registers[3] + 1;}
2  g_Registers[1] = GetWord(1); // e.g., IP_TOTAL_LEN is 256
3  for (g_Registers[0] = 0; g_Registers[0] < g_Registers[1]; g_Registers[0]++)
4  {
5    g_MemPublic.lBuffer[1]++;
6    for (g_Registers[2] = 8; g_Registers[2] > 0; g_Registers[2]--)
7    {
8      g_MemPublic.lBuffer[2]++;
9      g_Registers[3] = 1;
10     if (checkLoops() > 1000) return -1;
11   }
12   if (checkLoops() > 1000) return -1;
13 }
14 g_MemPublic.lBuffer[2]++;
15 for (g_Registers[3] = 0; g_Registers[3] < 10; g_Registers[3]++)
16 {
17   g_MemPublic.lBuffer[3]++;
18   g_Registers[5] = 1;
19   if (checkLoops() > 1000) return -1;
20 }
```

Listing 3.4: FOR loops in tailored C code for gcc.

Note that the FPL language hides most of the complexities of the underlying hardware. For instance, when using network processors, users need not worry about reading data into a μEngine's registers before accessing it. Similarly, accessing bytes in memory that is not byte addressable is handled automatically by the compiler.

In order to see some of the hidden details when compiling a FPL code for a network processor such as the IXP2400, we illustrate a piece of filter code in both languages: FPL and MicroC. Listing 3.5 shows a simple example of FPL code and Listing 3.6 shows the 'translated' code in MicroC language for Intel IXP2400 network processor.

```
1  MEM[2];                    // declares 2 shared memory locations
2  IF(PKT.B[23] == 17) THEN
3     M[0]++;                 // counts the UDP packets
4  FI;
5  IF(PKT.B[23] == 6) THEN
6     M[1]++;                 // counts the TCP packets
7  FI;
```

Listing 3.5: Example of FPL code.

Although the FPL example uses only two persistent memory location in order to count the TCP and UDP packets, we see in the MicroC equivalent code that for every memory access

there are a few more programming steps involved. For example, when reading or writing a shared memory location we need to pre-buffer the data through specific registers (transfer registers).

Listing 3.6 shows at line 4 the *read* transfer register declaration and then its usage for reading a DRAM memory location at line 7. Then, at lines 8-9, the read value is taken into local variables `uiFlowMem[]` (e.g., registers or fast local memory) for further computation as shown in lines 11-18. Similarly a read operation is also needed for a write as illustrated at line 5 where a write transfer register is declared, then at lines 20-21 the data is pre-buffered in the write transfer registers, and finally, at line 22 the value is written in the memory location `flow->smem`.

```
1  INLINE int filter_impl(UINT buf_index, __declspec(dram) buffer_t *flow)
2  {
3    __declspec(local_mem) __u32 uiFlowMem[2];
4    __declspec(dram_read_reg) __u32 drMemRead[2];
5    __declspec(dram_write_reg) __u32 drMemWrite[2];
6    SIGNAL_PAIR sig_dram;
7    dram_read(drMemRead, (__declspec(dram) void *)flow->smem, 1, sig_done,
         &sig_dram);
8    uiFlowMem[0] = drMemRead[0];
9    uiFlowMem[1] = drMemRead[1];
10
11   if (GetByte(buf_index, 23) == 17)
12   {
13     uiFlowMem[0]++;
14   }
15   if (GetByte(buf_index, 23) == 6)
16   {
17     uiFlowMem[1]++;
18   }
19
20   drMemWrite[0] = uiFlowMem[0];
21   drMemWrite[1] = uiFlowMem[1];
22   dram_write(drMemWrite, (__declspec(dram) void *)flow->smem, 1, sig_done,
         &sig_dram);
23   return 0;
24 }
```

Listing 3.6: Example of MicroC code for IXP2400 network processor.

We also note that in this simple example, in addition to the explained memory issues there are some specific functions used for access to the data in the packet fields. Such functions as GetByte, GetWord, GetDWord, are highly optimised for a quick data retrieval from the shared packet buffer *PBuf* and aware of specific data alignment. For instance, on the IXP2400, the DRAM memory locations are aligned on 8 byte chunks. In the example shown in Listing 3.6 at lines 11-18, `Getbyte` retrieves the packet field value located at a certain byte offset (e.g., 23 for IP_PROTO) in the specified packet index `buf_index`.

When deemed useful, however, users may *choose* to have some of the complexity exposed: it is, for instance, wise to declare additional arrays in fast hardware when that is available, like in the IXP2400's on-chip local memory instead of using external SRAM or DRAM.

3.1.3 The FPL-compiler architecture

Traditional compiler construction tools such as lex and yacc focus on the lexical analysis and parsing phases of compilation. However, they provide little support to semantic analysis and code generation. Building parse tree data structures and walking them is not terribly difficult, but it is time-consuming and error-prone [66].

In general, the basic steps in compilation are:

- Convert the program into an abstract syntax tree;
- Perform type-checking and semantic analysis on the tree;
- Rearrange the tree to perform optimisations;
- Convert the tree into the target code.

As observed in the compilation steps above, a large amount of work involves handling the abstract syntax tree (AST). There exist many implementations of AST manipulation tools to facilitate working with bison/yacc. We have chosen `TreeCC` [67] because of its high integration with bison and its easy to use manner.

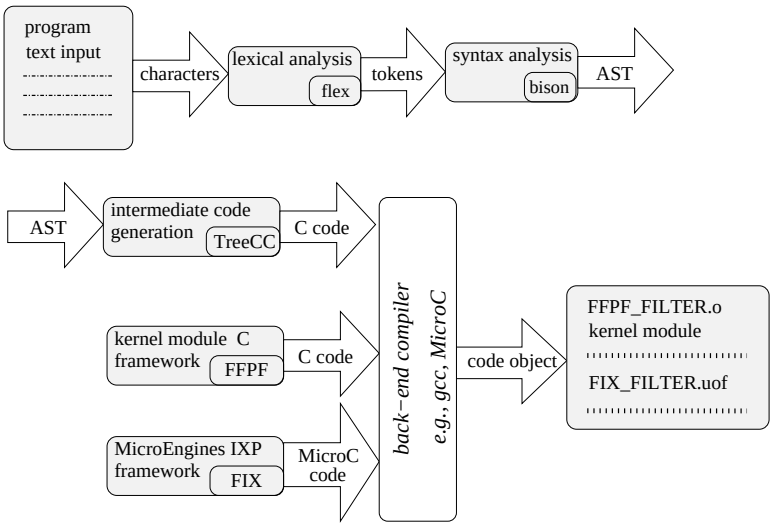


Figure 3.3: FPL-compiler architecture.

As shown in Figure 3.3, the FPL-compiler architecture consists of several modules each used for an intermediate processing phase of the input stream as explained below:

- The lexical phase (scanner) groups characters into lexical units or tokens. The input to the lexical phase is a character stream. The output is a stream of tokens. Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer). The scanner is implemented using `flex` and only the packet addressing part of the FPL-lex is shown, as example, in Listing 3.7.

```
1   LETTER    [a-zA-Z]
2   HEX       [a-fA-F0-9]
3   DIGIT     [0-9]
4   DIGITS    {DIGIT}{DIGIT}*
5   BYTE       "B"|"b"
6   WORD       "W"|"w"
7   DWORD     "DW"|"dw"
8   U1        "U1"|"u1"
9   U4        "U4"|"u4"
10  U8        "U8"|"u8"
11  U16      "U16"|"u16"
12  HI        "HI"|"hi"
13  LO        "LO"|"lo"
14  PKT_ADDR ("["{DIGITS}"]")
15  PKT_SUBFRACTION_OF_BYTE (("."){LO}|{HI}|({U1}"["[0-7]"]")|
        ({U4}"["[0-1]"]")))?
16  PKT_SUBFRACTION_OF_WORD (("."){LO}|{HI}|({U1}"["{DIGITS}"]")|
        ({U4}"["[0-3]"]")|({U8}"["[0-1]"]")))?
17  PKT_SUBFRACTION_OF_DWORD (("."){LO}|{HI}|({U1}"["{DIGITS}"]")|
        ({U4}"["{DIGITS}"]")|({U8}"["[0-3]"]")|({U16}"["[0-1]"])))?
18  PKT_ADDR_BYTE   (("."){BYTE}{PKT_ADDR}{PKT_SUBFRACTION_OF_BYTE}))
19  PKT_ADDR_WORD   (("."){WORD}{PKT_ADDR}{PKT_SUBFRACTION_OF_WORD}))
20  PKT_ADDR_DWORD (("."){DWORD}{PKT_ADDR}{PKT_SUBFRACTION_OF_DWORD}))
21
22  PKT_ID PKT({PKT_ADDR_BYTE}|{PKT_ADDR_WORD}|{PKT_ADDR_DWORD})
23  "R"           { return L_REG; }
24  "M"           { return MEM_SHARED; }
25  "MEM"         { return DECL_MEM; }
26  "REG"         { return DECL_REG; }
```

Listing 3.7: Part of lexical analysis (scanner) for FPL.

- The parser groups tokens into syntactical units. The output of the parser is a parse tree representation of the program. Context-free grammars are used to define the program structure recognized by a parser. The parser for FPL is implemented using `bison` and a part of it is shown, as example, in Listing 3.8.

```
1   oexpr: L_REG    '[' expr ']'    { $$ = lreg_create($3); }
2   | MEM_SHARED    '[' expr ']'    { $$ = memshared_create($3); }
3   | MEM_SCRATCH   '[' expr ']'    { $$ = memScratch_create($3); }
4   | PACKET '.' OFFSET_BYTE  '[' expr ']' { $$ = pktb_create($5);}
5   | PACKET '.' OFFSET_WORD  '[' expr ']' { $$ = pktw_create($5);}
6   | PACKET '.' OFFSET_DWORD '[' expr ']' { $$ = pktdw_create($5);}
7   statement:
8     oexpr ASSIGN expr        { $$ = assignTo_create($1, $3); }
9   | oexpr ASSIGN_PLUS expr   { $$ = assignTowithPlus_create($1, $3); }
10  | PRINT expr               { $$ = print_create($2); }
11  | expr                     { $$ = seqExpr_create($1); }
```

Listing 3.8: Part of syntax analysis for FPL.

We note that the `_create` constructs are specific to the TreeCC tool. Each '_create' construct (e.g., lreg_create) asks TreeCC to build a node with a specified type (e.g., lreg), using the specified parameters ($3), and insert the node into the abstract syntax tree (AST).

- The contextual analysis phase analyzes the parse tree for context-sensitive information. The output of the contextual analysis phase is an annotated parse tree. This step is implemented with the help of the `TreeCC` tool. An example of a type inferencing is illustrated in Listing 3.9.

```
1  /*!
2   * \brief Define the type code that is associated with a node
3   * in the syntax tree.
4   *  We use "error_type" to indicate a failure during type inferencing.
5   */
6  %enum type_code =
7  {
8    error_type,
9    int_type,
10   localreg_type,
11   localmem_type,
12   pkt_type
13 }
14 %node pkttype expression %abstract =
15 {
16   expression *expr;
17 }
18 %node pktb pkttype
19 %node pktw pkttype
20 %node pktdw pkttype
21 /*!
22  * \brief Define the "infer_type" operation as a non-virtual.
23  */
24 %operation void infer_type(expression *e)
25
26 infer_type(binary) {
27   infer_type(e->expr1);
28   infer_type(e->expr2);
29   if(e->expr1->type == error_type || e->expr2->type == error_type)
30   {
31     e->type = error_type;
32   }
33   else if(e->expr1->type == localreg_type || e->expr2->type ==
         localreg_type)
34   {
35     e->type = localreg_type;
36   }
37   else if(e->expr1->type == localmem_type || e->expr2->type ==
         localmem_type)
38   {
39     e->type = localmem_type;
40   }
41   else if(e->expr1->type == pkt_type || e->expr2->type == pkt_type)
42   {
43     e->type = pkt_type;
44   }
45   else
46   {
47     e->type = int_type;
48   }
49 }
50 infer_type(unary) {
```

```
51     infer_type(e->expr);
52     e->type = e->expr->type;
53   }
```

Listing 3.9: A type inference example used for FPL language.

- The code generator transforms the simplified annotated parse tree into `C` code using rules which denote the semantics of the source language. This step is also implemented with the help of `TreeCC` tool. An example of a code generator for `PKT.B[]` construct is illustrated in Listing 3.10. The example shows also how the code generator chooses which output format to use such as `gcc_k` (OS kernel for commodity PCs), `MEv2` for MicroC in IXP network processors.

```
 1    eval_expr(pkt),
 2    eval_expr(pktb),
 3    eval_expr(pktw),
 4    eval_expr(pktdw),
 5    {
 6     eval_value value;
 7     if (g_iOutputFormat == 1)
 8     {
 9       value = evaluationOfExpr_gcc_k(e);
10     }
11     else if (g_iOutputFormat == 2)
12     {
13       value = evaluationOfExpr_MEv1(e);
14     }
15     else if (g_iOutputFormat == 3)
16     {
17       evaluationOfExpr_MEv2(e);
18     }
19     else if (g_iOutputFormat == 4)
20     {
21       printf("Usermode-NOT yet supported!\n");
22     }
23     else
24     {
25       printf("Operator Evaluation: Unknown output format\n");
26     }
27     return value;
28    }
29
30    evaluationOfExpr_MEv2(pktb)
31    {
32     eval_value value;
33     infer_type(e->expr);
34     if (e->BoundChecking.eType == boundcheck_pre)
35     {
36       value = evaluationOfExpr_MEv2(e->expr);
37     }
38     else if (e->BoundChecking.eType == boundcheck_post)
39     {
40       sprintf(g_szTemp, "GetByte(buf_index, %s)",
             e->BoundChecking.szVarName);
41       strcat(g_szOutFileStream, g_szTemp);
```

```
42    }
43    else if (e->expr->type == int_type)
44    {
45      value = evaluationOfExpr_MEv2(e->expr);
46      sprintf(g_szTemp, "GetByte(buf_index, %ld)", value.int_value);
47      strcat(g_szOutFileStream, g_szTemp);
48    }
49    else if ((e->expr->type == localmem_type) || (e->expr->type ==
          localreg_type) || (e->expr->type == pkt_type))
50    {
51      printf("obsolete case for mem reg pkt type in pktb\nWork
            around: Use an assignment before if statement!\n");
52    }
53    return value;
54  }
```

Listing 3.10: Example of code generator for PKT.B construct in FPL.

3.1.4 The FPL-compiler tool

The FPL source-to-source compiler was designed to support multiple hardware architectures and hence, it generates code that will be consequently compiled by a back-end compiler for specific target hardware. Programs can therefore benefit from the advanced optimisers in the back-end compilers such as Intel μEngine C compiler for IXP devices and `gcc` for commodity PCs. As a result, the object code will be heavily optimised even though we did not write an optimiser ourselves.

Assuming that a filter expression written in FPL needs to run on two different hardware architectures (e.g., a commodity PC and an IXP network processor), the work-flow is illustrated in Figure 3.4 and is described below.

A first step consists of translating the filter into an intermediate FFPF module tailored to a specific hardware target ①. This module is a direct translation of FPL code into C code, including the compiler checks such as language constructs bound-checks, resource restriction checks, and forms the core of the future filter that FFPF needs to invoke on every packet. Then, the FPL-compiler passes the C code files to a back-end compiler according to a chosen hardware target Ⓐ, Ⓑ.

Choosing a network processor as hardware target, the work-flow follows the branch Ⓐ. In this case, the Intel MicroC compiler takes the translated code of the filter, compiles and links it to the FFPF on IXP framework (fix) ②. The result is a code object (e.g., fix.uof) that has to be loaded on the network processor. Next, the hardware needs to be started ③. The management operations (e.g., load/unload, start/stop) on the IXP network processors are performed with the help of *FIX management* tools.

When choosing a commodity PC as hardware target, the work-flow takes the branch Ⓑ. The translated code is compiled by the `gcc` compiler into a FFPF filter module ready to run in the Linux kernel ④. Next, the last step is loading the filter kernel module into the FFPF run-time environment by calling the FFPF framework's management function 'Insert filter' ⑧. The management operations such as loading/releasing of filters into/from the FFPF runtime are performed with the help of FFPF management tools as also described in detail in Section 4.1.2.

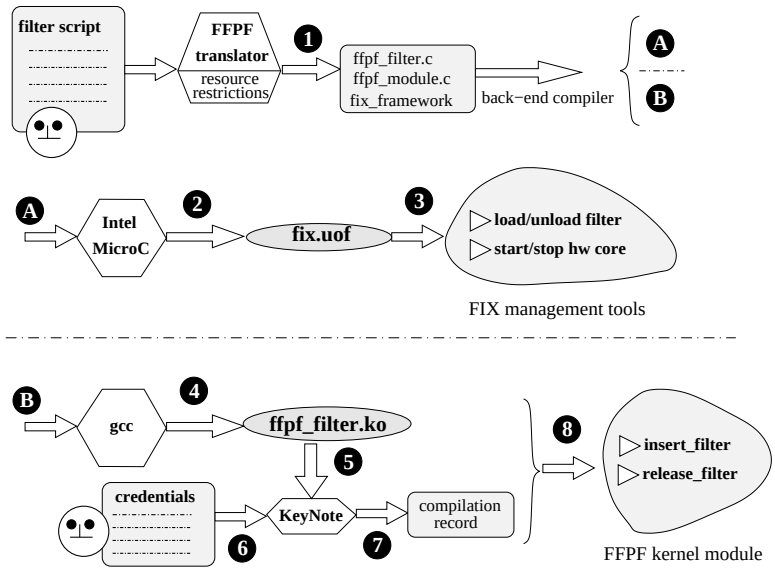


Figure 3.4: User compiles an FPL filter.

The FPL-compiler, in addition to the 'C translation', also makes some specific checks for security reasons. Because our framework allows users to run code in the lowest levels of the processing hierarchy, we must provide proofs of authorisation of the compiled code before injecting it into kernel or hardware. The FPL-compiler accomplishes this by using KeyNote to generate a *compilation record* ⑦, which proves that this module was generated by this (trusted) FPL-compiler [68]. The proof contains the MD5 of the object code ⑤ and is signed by KeyNote for certain user credentials ⑥ (e.g., userX can load code but only in a specific order, or only code without external function calls). When loading an FPL filter expression with a security demand, users provide the object code, as well as the code's compilation record. The code loader checks whether the code is indeed the FPL code generated by the local compiler and the credentials match its security policy, loads it in the kernel's FFPF target. The authorisation process is described in the next section.

3.1.5 Authorisation of FPL filters into the run-time environment

It is important to note that the use of FPL filters is not restricted to `root` users. In our framework, ordinary users may receive (in the form of credentials) explicit permission to monitor the network (possibly with restrictions, e.g., only packets with specific addresses). Whenever a user wants to insert a FPL compiled filter into the FFPF runtime, the FFPF admission control checks whether the user has the appropriate privileges to insert a filter with some specific combination of options applied to it.

Credentials and trust management used by our FPL-compiler and our run-time environment are implemented using the KeyNote [68] package. Our authorisation process provides fine-grained control that allows for complex policies. For instance, it is possible to specify such policies as: (1) a call to external function `strsearch` is permitted for packets arriving

```
KeyNote-Version: 2
   Comment: Within the application domain \ffp, the authoriser grants the
       licensee(s) the right to use a set of functions under specific
       conditions (e.g. regarding the number of occurrences and the order
       in which these functions are applied). Note: keys and signatures
       are abbreviated for clarity.
 Authorizer: "rsa-base64:MIGJAoGBAMbP4gS2x72ZF1PrnN//VEXfkYMtb="
 Licensees:  "rsa-base64:MIGJAoGBAKmynDiwN1tAKd6sGTHulfuyOoApl="
 Conditions: app_domain == "FFPF" && device_name = "eth0" &&
@$("PKT_COUNTER.num") < 2 && @$("STR_SEARCH.num") < 2 &&
@$("PKT_SAMPLER.num") > 0 && &$("PKT_SAMPLER.param.1") < 0.5 &&
@$("PKT_SAMPLER.first") < @$("STR_SEARCH.first") < 2 && -> "true";
 Signature: "sig-rsa-sha1-base64:i3zsmSamFnCs7SegUIPgJ6921In+U="
```

Listing 3.11: Example of credentials in Keynote.

on `eth0` *only* if it is preceded by a sampling function, (2) all calls to an external function `drop` *must* be followed by a return statement, and (3) filter x may only be connected to filter y, etc. These policies can only be checked if an entire chain of filters definition is available.

For example, one may want to specify that a user is allowed to apply a string search filter, but no more than one and only if the string search filter is applied after a sampling filter that grabs no more than 50 % of the packets. The assertions and credentials are processed using Keynote to check whether there are no conflicts between the user's credentials and the filter specifications. An example of the sort of assertions that may be expressed in the Keynote credentials with respect to this is shown in Listing 3.11. In the credential shown here, an authoriser grants a licensee (both identified by their public keys) the right to apply certain filters in a chain, subject to specific conditions. These conditions specify that the chain of filters is only valid in application domain FFPF, if the device that the chain is created for is one of eth0, if the chain has fewer than two packet counters, fewer than two string search operations and at least one sampler (with a specific first parameter), where the sampler should be applied before the string search (presumably to prevent a computationally intensive filter like string search from being applied to every packet). The credential is signed by the authoriser, e.g. the system administrator, or someone to whom the system administrator has delegated part of his/her powers (via another set of credentials).

The example shows that authorisation control guards against 'unsafe' filters, but can also be used to guard against 'silly' mistakes.

Authorisation control is optional. For instance, if the only party using FFPF is the system administrator, authorisation control may be left out to simplify management. We note that in the remainder of this thesis we focus on traffic monitoring rather than authorisation issues.

3.2 Evaluation of the FPL-compiler

The FFPF framework provides two distinct packet languages: FPL-1 and FPL (as described in Section 3.1). The FPL language has several advantages over its predecessor FPL-1. Firstly, it is compiled to fully optimised native code, rather than byte-code that is executed in an in-

terpreter. Secondly, it is based on a modern memory/registers/ALU model, rather than on the (slower) stack-based architecture used by FPL-1. Thirdly, FPL allows using of nested for-loops at the price of a few clock cycles spent on bound-checking. Fourthly, its similarity to traditional imperative programming languages makes FPL much more readable than its predecessor. The FPL approach was evaluated experimentally by implementing a set of programs in both FPL-1 and FPL and comparing their execution times. In addition, we also present the overhead of FPL comparing to the execution times of 'hand-crafted' C code of the same set of programs.

The benchmark consists of running the same filter expression (as illustrated in Listing 3.12), for a certain number (e.g., 15) of successive times, measuring the overhead, in clock-ticks, introduced by the filter check function for each time. The result is the median value of these 15 measurements and it is shown in Figure 3.5 among other filter expression processing results.

```
1  M[2]=10;                          //100; 250; 500 - maximum iterations number
2  M[0]=0;
3  FOR (M[1]=0; M[1]<M[2];M[1]++)
4     IF (PKT.B[M[1]] == 0x65)       // is this character 'A'?
5       M[0]++;                      // increment the counter
6     FI;
7  ROF;
```

Listing 3.12: FPL example.

This filter, in all its three versions (maximum iterations number differs), perform an extensive computation - searching of a specific character (e.g., 'A') in the packet data. When such a character is found, it is counted. In our benchmark, we made sure that all bytes in the packet had to be scanned.

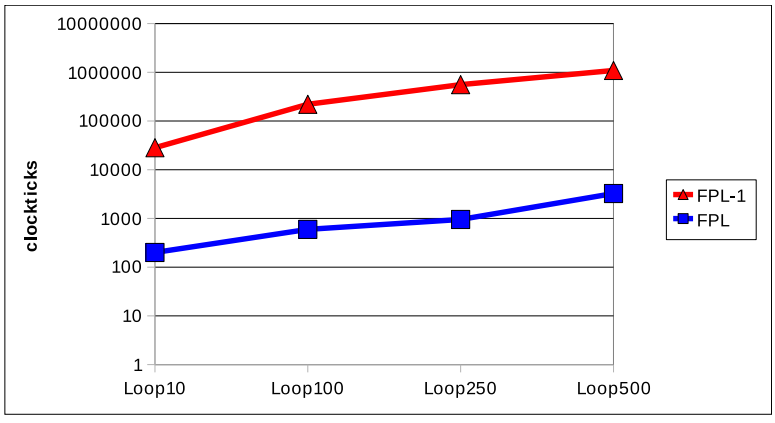


Figure 3.5: FPL-1 versus FPL.

It is clear that FPL easily outperforms FPL-1 (note that the scales are logarithmic). This is no surprise, as FPL-1 uses a handwritten interpreter, while FPL is fully optimised C code. Especially for more complex processing, such as looking at all bytes in the payload, this difference in performance is very big.

Figure 3.6 shows the overhead introduced by the FPL compiler for safety reasons such as bound-checks compared to the execution of straight C code. Note that the scales are linear and the overhead is about a few cycles which consists of 'hidden' if-statements to bound-check the for-loops execution as illustrated in Section 3.1.2.

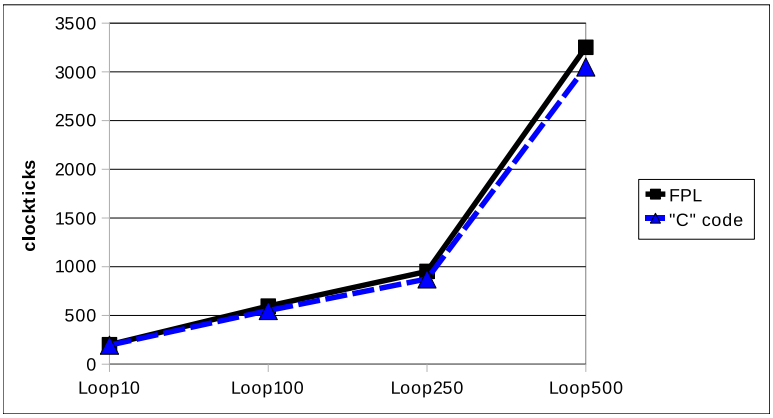


Figure 3.6: FPL versus "C" code.

	Assign	Hash
FPL-1	1020	1684
FPL	172	392
"C" code	168	388

Table 3.4: Comparison: FPL-1, FPL, and "C" code.

In Table 3.4 we also show a comparison of FPL-1 and FPL of the performance of applying a single assignment, and of a hash function. Again, it is clear that FPL is much more efficient, although it has a few cycles overhead compared to a C code due to the bound-checks on the input data used by `PKT[]` or the Hash operators. As a result, we believe that FPL is an important contribution to the FFPF framework.

3.3 Examples of FPL applications

In this section, we present two practical examples of simple packet processing applications written in the FPL language: (1) traffic characterisation in histograms and (2) traffic anonymisation.

3.3.1 Traffic characteristics histogram

This application is used for traffic analysis in a human readable form: histograms. The application does the following two checks on each incoming packet: (1) checks the packet size and classifies it within a certain range (e.g., 0-128 Bytes, 128-256 Bytes) and (2) checks

```
1  MEM[35];
2  REG[2];
3  R[0] = PKT.IP_TOTAL_LEN;
4  R[1] = EXTERN(GetPacketType, 0, 0); //0-UDP, 1-TCP, 2-ICMP, 3-Other
5  IF (R[0] < 128) THEN M[R[1]]++; FI;
6  IF (R[0]>= 128 && R[0]< 256) THEN M[R[1]+ 5]++; FI;
7  IF (R[0]>= 256 && R[0]< 512) THEN M[R[1]+10]++; FI;
8  IF (R[0]>= 512 && R[0]< 768) THEN M[R[1]+15]++; FI;
9  IF (R[0]>= 768 && R[0]<1024) THEN M[R[1]+20]++; FI;
10 IF (R[0]>=1024 && R[0]<1280) THEN M[R[1]+25]++; FI;
11 IF (R[0]>=1280 && R[0]<1512) THEN M[R[1]+30]++; FI;
```

Listing 3.13: FPL application for traffic characterisation.

the packet protocol and classifies it within one of the following class: UDP, TCP, ICMP, 'Other'. Then, the application counts the above packet characteristics into proper counters of a histogram such as illustrated in Figure 3.7.

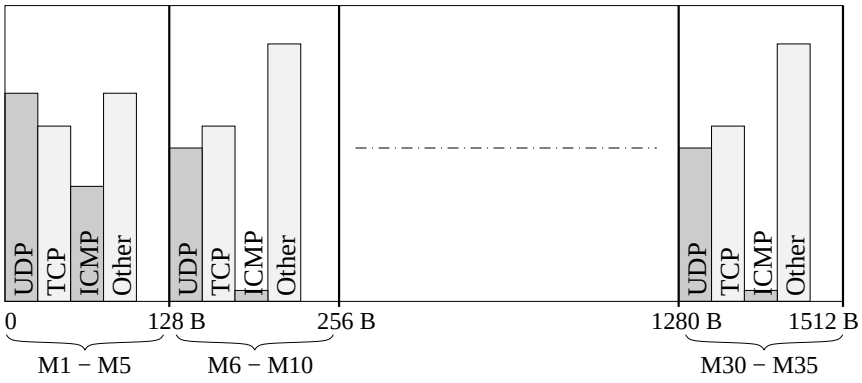


Figure 3.7: Traffic histogram.

The application written in FPL language is shown in Listing 3.13. It uses the MEM buffer to store persistent data: the counters. We partition the buffer as follows: seven packet size ranges in which we may have four different packet types. Note that for the sake of code simplicity we allocated five counters per range though we use only four: 0-UDP, 1-TCP, 2-ICMP, and 3-Other. We also observe the use of 'GetPacketType' external function which is separately shown in Listing 3.15. Although the 'GetPacketType' is small, simple, and could easily be written within the application, we made it an external function for the purpose of illustrating code re-usage between two applications: the current histogram and the 'packet anonymisation' as presented in the next section.

3.3.2 Packet anonymisation for further recording

In this application we address the problem of saving on external memory (e.g., hard-disks) real-time traffic for offline analysis. An important requirement is to provide privacy on the

'tapped' traffic. For example, sensitive data such as http payloads that may include private data must not be stored. The process of hiding the private data from a packet is also called 'anonymisation'. We propose a simple application that hashes the payload and pushes the IP, TCP or UDP headers together in a shared memory available for further storage. Note that the retrieving of the stored packet data from shared memory and storing them on external support (e.g., hard disks) is left out of scope of this experiment. If the packet is of other type than TCP or UDP, then it saves the entire IP header.

The application written in FPL language is shown in Listing 3.14.

```
1  MEM[4]; // keep indexes for dumping packet in the big shared MEM
2  REG[2];
3  R[0] = PKT.IP_TOTAL_LEN;
4  R[1] = EXTERN(GetPacketType, 0, 0); //0-UDP, 1-TCP, 2-ICMP, 3-Other
5  IF (R[1] == 0) THEN // UDP packet
6    M[0] = HASH(20, R[1], 4096); //Hash from Start_UDP till end of PKT
7    EXTERN(SavePktDataUDP, 0, 0); // Save the UDP_hdr + hash_value (at M[0]),
         starting at specific offset (M[1])
8    M[1]++;
9  ELSE
10   IF (R[1] == 1) THEN // TCP packet
11     M[0] = HASH(20, R[1], 4096); //Hash from Start_TCP till end of PKT
12     EXTERN(SavePktDataTCP, 0, 0); // Save the TCP_hdr + hash_value (at
           M[0]), starting at specific offset (M[2])
13     M[2]++;
14   ELSE // IP packet
15     M[0] = HASH(0, R[1], 4096); // Hash from Start_IP till end of PKT
16     EXTERN(SavePktDataIP, 0, 0); // Save the IP_hdr + hash_value (at M[0]),
           starting at specific offset (M[3])
17     M[3]++;
18   FI;
19 FI;
```

Listing 3.14: FPL application for packet anonymisation.

```
1  MEM[1];
2  M[0] = PKT.IP_PROTO;
3  IF(M[0] == UDP) THEN
4    RETURN 0;
5  ELSE
6    IF(M[0] == TCP) THEN
7      RETURN 1;
8    ELSE
9      IF(M[0] == ICMP) THEN
10       RETURN 2;
11     ELSE
12       RETURN 3;
13     FI;
14   FI;
15 FI;
```

Listing 3.15: FPL external function: GetPacketType.

3.4 Summary

In this chapter we saw the FPL language we use to provide a common user interface for building traffic processing applications regardless of the hardware support that lies beneath. In addition to the performance of the compiled code, our FPL language achieves important requirements of the main FFPF packet processing framework such as safety, easy programmable, and hiding hardware specific details.

In the next chapter, we will consider implementations of FPL on specific target platforms.

Chapter 4

FPL Run-time Environments

In this chapter we introduce the run-time extensions made to an existing packet processing framework, Fairly Fast Packet Filter (FFPF), in order to support the previously introduced FPL applications onto several hardware architectures in use these days: commodity PCs, network processors, and FPGAs.

While in the next chapter we will show how we can handle even faster links by distributing the processing over multiple nodes, this chapter assumes a single node processing environment.

4.1 FFPF on commodity PCs

The first FFPF implementation was built on commodity PCs using Linux. It consists of a buffer management system implemented in a kernel module, various APIs for userspace support (e.g., pcap libraries), a compiler for the FPL language, and a management toolkit for easy integration of development and debugging tools.

4.1.1 Buffer management

The main purpose of the buffer management system (BMS) is to capture all packets that are marked as interesting by a low-level processing task (e.g., a kernel FPL filter) and hand a reference to these packets to the appropriate user applications.

Note that in our packet processing framework, FFPF, there is always one producer (packet source) and many consumers (filters), which allows us to implement lock-free circular buffers to store the packets.

As also illustrated in Figure 4.1, Circular buffers in FFPF have two indices to indicate the current read and write positions. These are known as R and W, respectively. Whenever W catches up with R, the buffer is full. The way in which the system handles this case is defined by the buffer management system (BMS). The modular design of FFPF allows different BMSs to be used. The administrator chooses the BMS at startup time. The optimal choice depends on the mix of applications that will be executed and their relative importance.

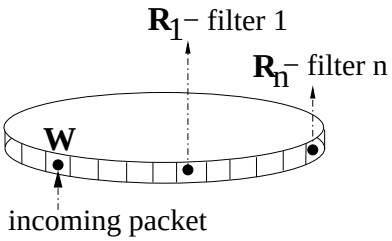


Figure 4.1: Circular buffer in FFPF.

At the time of writing, two BMSs have been defined. The first is known as 'slow reader preference' (SRP) and is commonly used in existing systems. The second is known as 'fast reader preference' (FRP) and is a novel way of ensuring that fast readers are not slowed down by tardy applications.

Buffer management is concerned with how readers and writers synchronise and share their buffers. SRP corresponds to 'traditional' buffer management in which new packets are dropped if the buffer is full. While convenient, the disadvantage is that one tardy application that fails to read the (shared) *PBuf* at a sufficiently high rate, causes packet loss for the entire group. FRP, in contrast, simply keeps writing regardless of whether the buffer is full. The trick is that it enables applications to check *a posteriori* whether the packets they just processed have been overwritten. For efficiency, applications in both SRP and FRP may process their packets in *batches* (e.g. 1000 packets at a time) in order to minimise context switches. Details about SRP and FRP are provided in [14].

A high-level overview of the FFPF implementation on commodity PCs is shown in Figure 4.2. It shows that the central component of FFPF is the Buffer Management System (BMS). BMS consists of a main buffer (*PBuf*) shared by all applications that may access it. BMS also includes several secondary buffers and pointer lists needed to assure a good system management. For instance, assuming the system has *m* filters ($F_1, F_2, ...F_m$) registered in the filter list `g_listFilters`. Each filter may access a packet in the shared *PBuf* by one of the following two ways: (1) directly getting the next available packet in *PBuf* by reading X up to the global W position, increasing its local R position in *PBuf*, and eventually marking, into a local index buffer (*IBuf*), the packets found interesting for other filters such as higher level applications, or (2) by getting the next packet pointed by the local *IBuf* being marked as interested beforehand by other filters. The latter option is useful when using a chain of filters.

The main purpose of the BMS is to capture all packets that are considered 'interesting' and hand a reference to these packets to the appropriate applications. The idea behind FFPF is simple. Users start *applications* in user-space (see App.1, App.n in Figure 4.2) that load *filters*. A filter is an application oriented on packet processing only and it runs at low levels such as OS kernels, or even in hardware. An application may use the processing results of one or more loaded filters by way of a shared memory that is called *MBuf* and is located inside each filter. Moreover, an application may use a filter as a 'pre-processor' so as to offload part of its heavy packet processing job onto lower processing levels. In this case, the application retrieves from its loaded filter indexes to the packets needed for a more exhaustive processing than the filter was able to perform.

FFPF framework provides the basic receiving functionality. When a packet arrives, FFPF

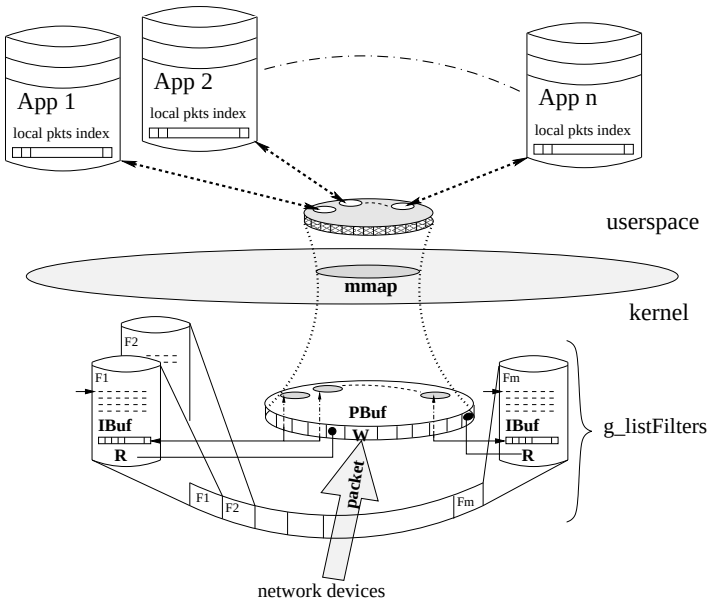


Figure 4.2: The FFPF implementation on commodity PCs.

pushes it into the main shared packet buffer *PBuf* and advances the global W pointer with one slot. When a filter classifies a packet as 'interesting' for other filters or user applications, it returns a non-zero value, a pointer to the packet is placed in the filter's index buffer *IBuf*, and the local R pointer advances one slot. An application may use the filter's index buffer to find the packets in which it is interested. In addition, a filter has its own chunk of memory (known as `'MEM'`) that is shared with the application and which it may use to store results to be read by the application or temporary values that do not disappear between invocations (persistent state). All buffers are memory mapped, so no copying between kernel and user-space is necessary.

4.1.2 The FFPF run-time environment for compiled filter object

The interface between FFPF and the compiled filter object deals with two aspects: inserting/removing of filter and run-time handling of the registered filters. In order to provide such functionalities, we extended the FFPF framework with a management toolkit. The FFPF toolkit includes tools such as a 'loader' for filter loading, a 'manager' for hardware cores, etc. This toolkit may use different tools or ports of the same tool for different hardware targets. For instance, the *loader* on commodity PCs is a simple kernel module loader, while on a network processor, the loader deals with transferring the filter object onto the hardware, loading and starting the hardware remotely, etc.

Assuming that a filter has been compiled in the filter kernel module as it targets a commodity PC, it must be inserted and registered by FFPF. The user can simply inject the filter module into the kernel by using FFPF loader (e.g., '`./ffpf_loader filter_nn.ko`'), as shown in Figure 4.3 ①. While doing so, FFPF checks also the module authorisation and

decides whether the module is going to be accepted or not.

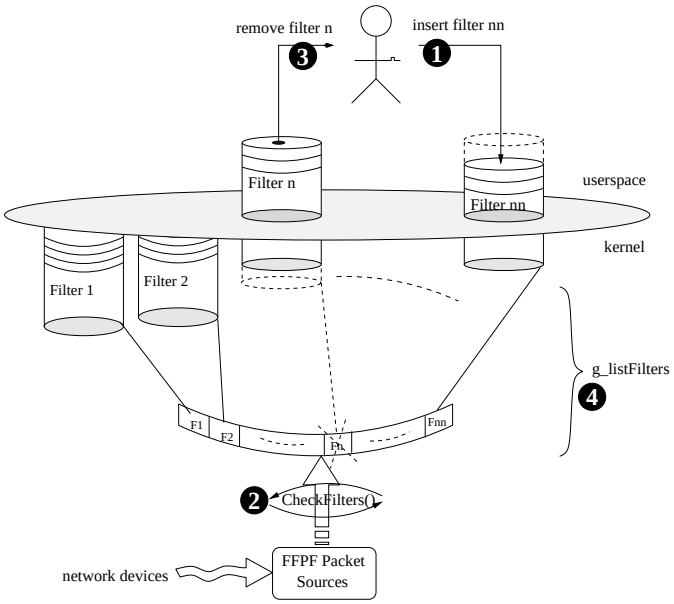


Figure 4.3: FFPF filters interfaces.

In commodity PCs, the FFPF "packet source" employs a hook registered within Linux ②, as also described in the next section. The hook behaves almost as an interrupt handler, and is invoked at each incoming packet. The hook function gets as parameter a pointer to the current packet. The packet pointer is passed to each filter registered into the filters list `'g_listFilters'` ④. The filter list is maintained by the FFPF core in such way that any filter registering/unregistering operation goes safely through only one point and thus mutual exclusion is ensured.

The user can release a packet filter from the filters list by invoking the FFPF unloader on the specific filter (e.g. '`./ffpf_loader -u ffpf_filter_n.ko`') as illustrated in Figure 4.3 ③. When removing the module, FFPF automatically checks whether the specified filter is registered or not and if so, releases the pointer from '`g_listFilters`'.

4.1.3 FFPF packet sources

Packets enter the FFPF framework via a call to an FFPF function called *hook_handle_packet()* which takes a packet as argument. As this is the only interface between the code responsible for packet capture and the FFPF packet handling module, it is easy to add new packet sources. Currently, three sources are implemented.

The first source, known as `netfilter`, captures packets from a netfilter hook. Netfilter is an efficient abstraction for packet processing in Linux kernels (from version 2.4 onward). The second source, known as `netif_rx`, is a lower level routine that also works with older kernels. The third packet source, known as `ixp`, differs from the other two in that the net-

work processor device (e.g., IXP1200, IXP2400, IXP2850) is assumed to be dedicated to monitoring in the FFPF framework which means that part of the processing can be offloaded to it. As this packet source is a substantial project in and of itself, we will describe its main characteristics in the next sections.

4.2 FFPF on NPs: NIC-FIX

Packet handling in modern workstations works well for slow, sub-gigabit speeds but fails at higher rates. While some operating systems fare a little better than others, this statement is true irrespective of one's choice of operating system [69]. The problem is caused by a combination of hardware and software bottlenecks.

Memory and peripheral bus technologies struggle to keep up with backbone link rates. Even if a workstation does manage to get all packets across the bus in host memory and from memory in the CPU, inefficient packet handling by the OS still makes it difficult to process packets at high speeds. The problems are commonly rooted in the overhead of interrupt handling, context switching and packet copying [70]. As it stands, we conclude that common workstations with current hardware and software configurations are not suitable for high-speed network monitoring. At the same time, the need for affordable network monitors is growing, e.g., for security, traffic engineering, SLA monitoring, charging, and other purposes. In this section we present NIC-FIX, an extension of the Fairly Fast Packet Filter (FFPF) [14] network monitoring architecture on network cards with Intel IXP network processors (NP) [47,48]. The extended FFPF architecture can be described as 'bottom-up' in that packets are handled at the lowest processing level and few packets percolate to higher levels, as also illustrated in Figure 4.4. Moreover, higher levels only take action when prompted to do so by the lower layers. This is a known approach, e.g., in router design [71].

NIC-FIX extends the FFPF processing hierarchy

In the extended FFPF, NIC-FIX forms the lowest level of the processing hierarchy, as shown in Figure 4.4. NIC-FIX allows us to deal with common hardware bottlenecks by offloading packet processing to the network card when possible, thus reducing strain on the memory and peripheral buses.

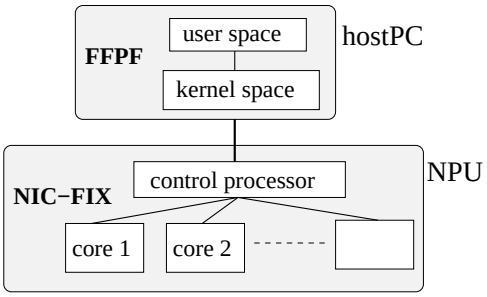


Figure 4.4: A processing hierarchy in FFPF.

The control processor

At the present time, NPs make use of a general CPU, such as ARM or XScale on Intel IXP, in order to control their complex parallel architecture: hardware cores, buses, different memory types, etc. We run embedded Linux (a big-endian port of Linux for ARM CPUs) on the control processor. In addition, we designed and developed a management toolkit that contains components responsible for control tasks such as initialization, loading and control of cores, memory mapping of buffers from SDRAM to the host, initialization of different memory buffers, etc.

The hardware cores

Processing packets at high speed line rates (multi-gigabits/sec) is difficult to accomplish using general purpose processors because of the growing gap between the link and the memory access speeds, as also described in Section 1.3.2.

We use Intel IXP network processors. An IXP makes use of multiple hardware cores, namely (μEngines), in order to provide enough parallelism and, consequently, to be able to cope with high speeds (multi-gigabits/sec). As also described in Section 2.2, the μEngines are RISC processors specifically designed for packet processing. For instance, a μEngine has no floating point and does not run any operating system. μEngines are interconnected to each other and to the outside world (e.g., memory, Ethernet transceivers) through multiple hardware interfaces that may run in parallel: buses, signals, memories, registers, etc. μEngines are under the control of a general purpose processor: the control processor.

4.2.1 Mapping packet processing tasks onto the NP hardware

The speed advantage of using NPs as hardware platform for packet processing depends mostly on the way the processing tasks can run in parallel. Supposing we have an NP architecture that has built-in multiple hardware cores (see the example of NPs architecture details in Section 2.2), then we need to map a user application (e.g., packet processing filters) to a hardware platform in order to benefit from the parallel hardware features as much as possible.

Often, a packet processing application consists of multiple processing tasks (e.g., to count TCP flow activity, and scan for worms in UDP packets). The NP architecture provides different possibilities for the granularity at which parallelism can be implemented: at core level (μEngines), or even at thread level (multiple hardware supported threads per core). For the sake of simplicity, we chose to map one task per hardware core although we could have also opted for a thread granularity. In other words, we partition the NP at μEngine granularity and therefore, an application composed of multiple tasks would use multiple μEngines. The idea of mapping one to one (one task per μEngine) was also used recently in a network device virtualisation by Kumar *et al.* [72]. Merging results of individual tasks happens at a higher level in the processing hierarchy (e.g., the general purpose processor of a host PC).

Our architecture supports three mapping options: (1) one task may process all packets, (2) different tasks may process the same packet, and (3) different tasks, or a same task mapped onto several hardware cores, may process different packets (load balance). These aspects are illustrated in Figures 4.5.a, 4.5.b, and 4.5.c, respectively.

As shown in Figure 4.5.a, any packet processing application has, at a minimum, the following two tasks: ① a receiver (`Rx`) that retrieves all the incoming packets from the network interface and stores them into a local shared buffer for further processing; ② a custom packet processing task (`T`) that takes a subset of all available packets in the shared buffer, performs some packet processing, and stores the processing results in shared buffers to a higher level module (user's control application). The task `T` runs on a μEngine, and may process packets one by one, using one thread, or it may process multiple packets in parallel using multiple threads running on the same μEngine (every thread processes one packet).

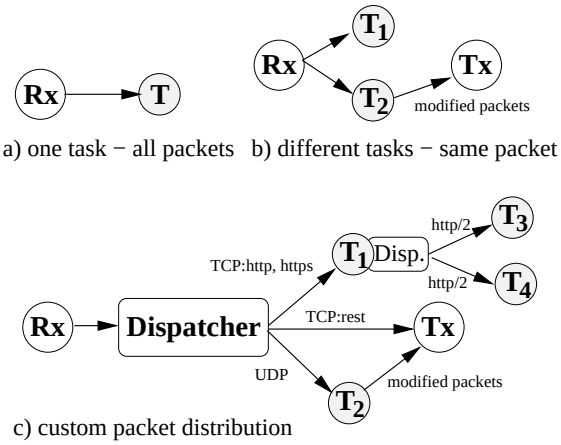


Figure 4.5: Mapping packet processing tasks onto NP's hardware cores.

Figure 4.5.b shows an example when different tasks may process the same packet. For instance, a simple application that has two tasks (T_1, T_2), both processing all the received packets, but T_1 does packet processing and then drops the packets (e.g., TCP flow accounting) and T_2 modifies the received packets and then sends them to a transmitter (`Tx`).

Figure 4.5.c shows an example where different tasks may process different packets: a load balancing situation, for instance. In other words, this application type requires processing of different packets in different tasks and hence, the application needs to split the incoming traffic in sub-streams. Figure 4.5.c illustrates such an example of custom packet distribution. Supposing that tasks T_3 and T_4 process only part of the received http traffic (e.g., load balance the web stream), part of the received traffic (e.g., non-web TCP packets) is sent out, and task T_2 also sends out some of its modified UDP packets. This custom packet distribution over the processing tasks is implemented by a `dispatcher` that consists of another packet processing task that mainly does light processing: checking of some of the packet header fields and then enqueue the packet to the proper task according to the programmed classification.

A dispatcher may be embedded in a basic task such as `Rx` or in a custom processing task such as T_1. Moreover, a dispatcher may be also a separate task, fully programmable by the user like any other task, interconnected within a processing hierarchy. The latter dispatcher type helps when re-programming of the dispatcher is required to be done 'on-the-fly' by not affecting the other processing tasks and avoid packet loss.

In the following sections we continue to present our NIC-FIX design and then some

specific software implementation on the first three Intel network processor generations: the IXP1200, IXP24xx, and IXP28xx.

4.2.2 NIC-FIX architecture

We use the packet processing at high speeds as introduced by FFPF and presented in Section 2.1 (e.g., minimising the packet copy) and exploit the hardware features (e.g., parallelism) of a network processor such as the Intel IXP presented in Section 2.2 in order to achieve the FFPF stated goals: scalability (in number of applications), flexibility, and high speeds, but at line rates (multi-gigabits/sec).

Figure 4.6 shows the FFPF design on the IXP NPs. The design has two aspects: the control and data planes. The data plane involves software running on the NP hardware cores (μEngines) that perform packet processing. The data plane consists of ④ a buffer management system, similarly to the FFPF implementation on commodity PCs, and two types of software tasks: basic tasks such as ① `Rx` and ② `Tx` for receiving and transmitting of packets, respectively, and ③ custom tasks (the user application). While the basic tasks are provided by the NIC-FIX framework, the custom tasks are part of the user applications written in the FPL language and compiled for IXP hardware.

The control plane uses multiple software modules (e.g., client/server) that run on the NP ⑤, and on a host_PC ⑧ in order to provide low-level functionalities (hardware initialisation, code upload, etc.) needed by any user application at a high-level.

Packet reception, processing, and transmission

As shown in Figure 4.6, NIC-FIX uses the first μEngine (μE_0) for a gigabit receiver ①. The receiver stores each incoming packet into a circular buffer in DRAM called *PBuf* ④ which is shared across all μEngines. The last μEngine is allocated to the gigabit transmitter ②. The transmitter pulls each enqueued packet (using its own index buffer – *TBuf*) from the main *PBuf* and transmits it out to a gigabit port. The intermediate μEngines are ready to use for custom packet processing in a fully programmable manner (e.g., in a chain or in parallel), as described earlier in Section 4.2.1. NIC-FIX feeds each processing task with indexes to the available packets (in *PBuf*). Next, every packet processing task reads partly or completely the available packet stored in *PBuf*, processes it, and classifies it either for further processing (by enqueuing the packet index into its *IBuf* or into the *IBuf* of the next task in a chain), or for transmission out to a certain port (by enqueuing the packet index into *TBuf*). A packet is enqueued in its *IBuf* when the high-level components of the FFPF framework (e.g., kernelspace, userspace filters) are interested in it. As a result, much of the computation can be offloaded to the hardware accelerated filters.

Buffer management

One of the critical issues in the FFPF framework is data copying. Similar to the FFPF implementation on a commodity PC (see Section 4.1.1), there is a shared circular packet buffer (*PBuf*) in NIC-FIX ④. The purpose of such a novel buffer is to reduce the data copying needs. *PBuf* is shared by all μEngines, the XScale core and even higher layers through memory mapping. Whenever a packet is used by one or more 'consumers' (μEngines, XScale,

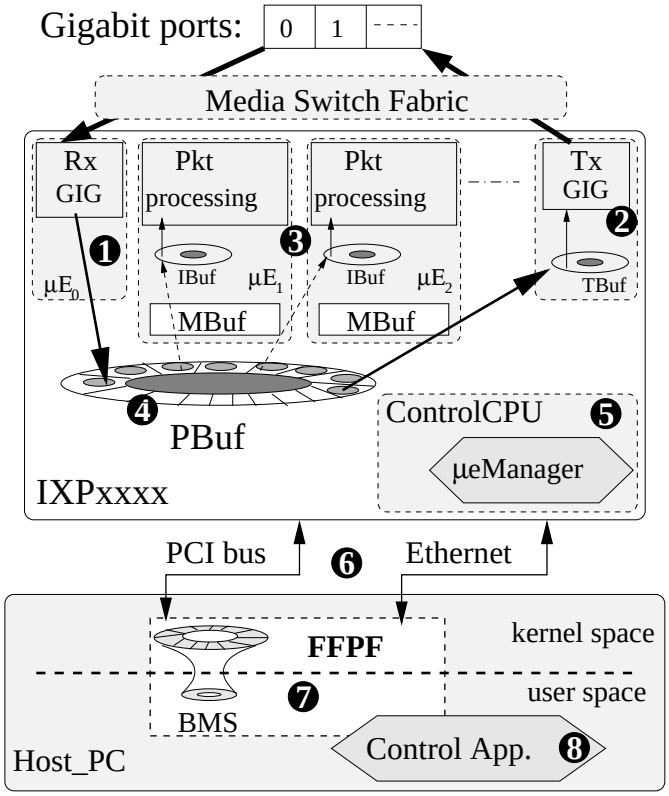


Figure 4.6: NIC-FIX architecture.

etc.), it is just placed in *PBuf* once. In other words, rather than copying a packet to each individual consumer, it is copied at most once. Therefore, the buffer has only one writer and several readers. The writer's job is quite simple: it increases the write index of *PBuf* by one as every packet is saved into the buffer. Each reader has its own index buffer (*IBuf*). The *IBuf* of reader X stores a number of indexes pointing to the corresponding packets in the *PBuf* in which X is interested. A reader's job is to check whether its index is not bigger than the write index to avoid 'over-read' and it stops the processing when the indexes are equal (no more packets available in *PBuf*).

A subtle issue concerns the size of the read and write indices, when we need to deal with the possibility of overflow caused by the continuously increasing index. A 32 bit integer can index four giga packets. At gigabit link speeds, it takes only a few minutes to overflow the index. On the other hand, our hardware architecture provides atomic read/write operations only for the memory that is aligned to 4 Bytes boundaries. Comparing all pros and cons we still use the 32 bit index. We count indexes modulo 4 giga as a solution for an overflow problem.

In addition to the shared packet buffer, a filter has its own chunk of memory (known as

MBuf) that is shared with the application and which it may use to store results to be read by the application or temporary values that do not disappear between invocations (persistent state).

Resource management

As illustrated in Figure 4.6, the control path is composed of a client/server application that interconnects a hostPC with our IXPxxxx system over various media interfaces (e.g., PCI bus, Ethernet). The control application runs its server on the host_PC ⑦, and the clients on the control processor of each connected NP ④. The control application was designed to support multiple interfaces such as PCI bus or Ethernet (10/100 Mbps) ⑤. In other words, we can control the IXPxxxx NP in a PCI hosted variant as well as in a remote connection over the management ethernet port. The client running on the control processor of each NP, called μeManager, is responsible, in addition to the communication function, for the low-level control of the hardware such as initialisation, code loading, hardware core start/stop, etc.

In this section, we presented our NIC-FIX architecture design that currently supports the first three Intel IXP network processor generations available to date: the IXP1200, IXP2400, and IXP2850. However, each NP has some specific features that change from one generation to another. In the following sections, we shortly describe the specific NIC-FIX implementations on each of the IXP generation.

4.2.3 NIC-FIX on the IXP1200

The IXP1200 runs software in two areas of the network processor: (1) six μEngines with four hardware threads each, providing 24 parallel threads for packet processing, and (2) the control processor (a StrongARM core) with chip/board support package for Linux and VxWorks, drivers, communications and some basic network applications.

As described in the previous section, in NIC-FIX, an FFPF filter is mapped onto a μEngine. In addition, NIC-FIX uses two μEngines for basic filters (`Rx`, `Tx`). Given that the available number of μEngines on the IXP1200 is six, as a consequence, the IXP1200 can support a maximum of four filters. A filter uses all four threads, hardware supported by each μEngine, to process the packets one by one. If the filter determines that the packet is interesting also for other filters in the processing hierarchy, the μEngine places an index for the packet in the filter's *IBuf*.

As the IXP1200 is considered 'obsolete' and no longer supported by Intel, newer IXP generations are available on the market: IXP2400, IXP2850. Because the newer versions of the IXP support more μEngines at higher clock-rates, both the number of filters that can be supported and their speeds increase in the NIC-FIX implementations on the next IXP generations.

4.2.4 NIC-FIX on the IXP2400

Similar to the first IXP generation, IXP1200, this generation also uses a dual-plane design: control and data planes. The control plane uses an XScale processor that is ARMv5 compliant, and it is more powerful in both, clock-rates and also instructions set, than that of the

previous IXP. The data plane is composed of eight μEngines running at higher clock-rates than those of the IXP1200. Despite the clock speed growing, there are other improvements in this generation such as a new and fast on-chip memory (`localmem`), direct signals between μEngines (e.g., next-neighbour), more memory transfer registers, etc. The FPL compiler was enhanced to support some of the new generation's features. For example, the `'MEM'` syntax on improved architectures such as the IXP2400 or IXP2850 is implemented on `localmem` memory type instead of slower off-chip SRAM like in the IXP1200.

4.2.5 NIC-FIX on the IXP2850

The IXP2850 NP has both planes, XScale core for control plane and μEngines for data plane, compatible to the IXP2400. Therefore, our FFPF implementation on the IXP2850 uses the same design discussed before. However, we used a dual-processor development platform (IXDP2850) and the specific implementation details can be found in this thesis' case study (Chapter 8.3.2).

4.3 FFPF on FPGA: NIC-FLEX

We have argued that parallelism can be exploited to deal with processing at high speeds. As we have shown, a network processor (NP) is a device specifically designed for packet processing at high speeds by sharing the workload between a number of independent RISC processors. However, for very demanding applications (e.g., payload scanning for worm signatures) more power is needed than any one processor can offer. For reasons of cost-efficiency it is infeasible to develop NPs that can cope with backbone link rates for such applications. An attractive alternative is to use a reconfigurable platform such as an FPGA that exploits parallelism at a fine granularity.

Figure 4.7 illustrates the advantages and disadvantages of the most used architectures in packet processing: ① commodity PC, ② network processor (NP), and ③ hardware reconfigurable (FPGA). Although the packet processing speed grows when the parallelism increases, at the same time, the programmability becomes harder.

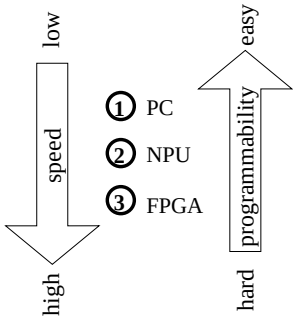


Figure 4.7: Speed versus programmability in three architectures used in packet processing.

As illustrated in Figure 4.8.a, in a commodity PC architecture, the incoming packets need

to be transferred across (slow) buses to the host CPU memory. Then the CPU reads, processes, and writes back the packets to the memory. Finally, the processed packets may be sent out. All these steps make it difficult to process packets (e.g., scan for worms signatures) at high speeds (multi-gigabits/sec) because of the growing gap between the link and the memory access speeds, as we argued in Section 1.3.2.

We have previously shown how the FFPF monitoring framework handles high speeds by pushing as much of the work as possible to the lowest levels of the processing stack (see Figure 4.8.b). The NIC-FIX architecture [73] showed how this monitoring framework could be extended all the way down to the network card. To support such an extensible programmable environment, we introduced the special purpose language known as FPL.

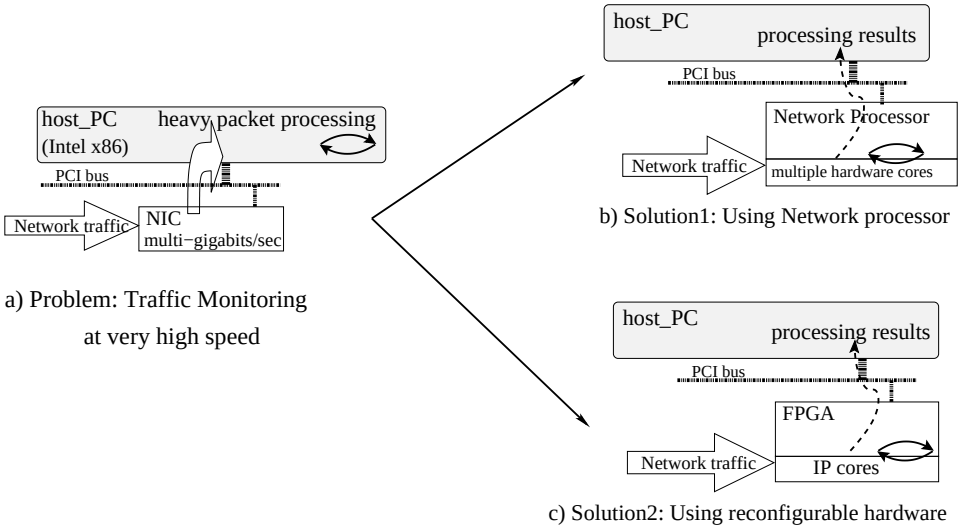


Figure 4.8: Moving to special purpose embedded systems.

In this section, we exploit packet processing parallelism at the level of individual processing units (FPGA cores) to build a monitoring architecture: NIC-FLEX (see Figure 4.8.c). Incoming traffic is stored in fast off-chip memory, wherefrom it is processed by multiple FPGA cores in parallel. The processing results are first stored in a very fast local memory and then passed, on demand, to higher levels (e.g., user space tools). The main contribution of this section consists of extensions to our FPL language that explicitly facilitates parallelization of complex packet processing tasks. Also, with NIC-FLEX we extend the FFPF architecture with specific packet processing support to create a flexible and fast filtering platform. Experiments show NIC-FLEX to be able to handle complex tasks at gigabit line-rate.

NIC-FLEX builds on the idea of extensible system-on-programmable-chips that was advocated by Lockwood *et al.* in [74] for firewalling. However, we use it to provide a generic high-speed packet processing environment by using the Compaan/Laura tool chain [75, 76] that automatically transforms user code into synthesizable VHDL code that targets a specific FPGA platform.

4.3.1 High-level overview

At present, high speed network packet processing solutions need to be based on special purpose hardware such as dedicated ASIC boards or network processors (see Figure 4.8.b). Although faster than commodity hardware (see Figure 4.8.a), solutions based even on these platforms are surpassed by the advances in reconfigurable hardware systems like FPGAs for certain applications (e.g., those requiring encryption/decryption).

To exploits this trend we propose the solution shown in Figure 4.8.c, which consists of mapping the user's program onto hardware, processing the incoming traffic efficiently, and then passing the processing results back to the user.

The software architecture is comprised of three main components, as illustrated in Figure 4.9. The first component ① is a high level interface to the user and kernel space of an operating system (e.g., Linux) and is based on the Fairly Fast Packet Filter (FFPF) [14] framework. The second component ② is the FPL-compiler that takes a program written in the FPL packet processing language ⓐ and generates a code object ⓑ for the lowest level of processing: reconfigurable hardware. The third component ③ is a synthesiser tool that maps specific processing algorithms (e.g., Aho-Corasick) onto an FPGA platform and makes use of a tool chain (Compaan/Laura) that automatically transforms sequential code into parallelised hardware code.

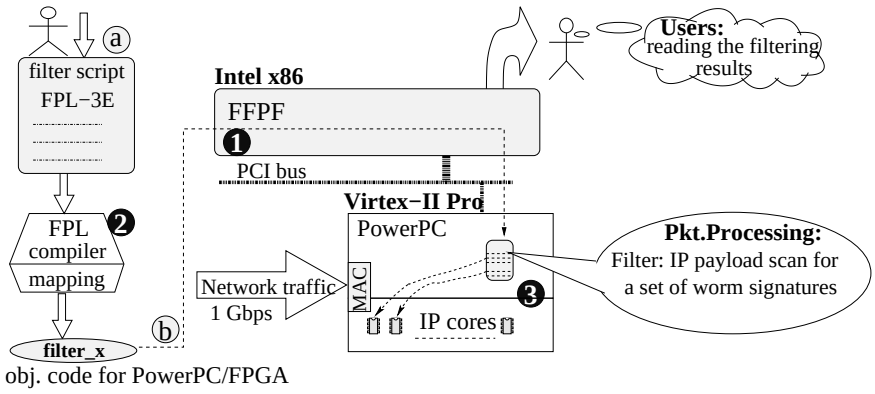


Figure 4.9: NIC-FLEX packet processing architecture.

4.3.2 Extensions to the FPL language

As our architectural design relies on explicit hardware support, we needed to introduce this functionality into our framework. With FPL, we adopted a language-based approach, following our earlier experiences in this field. We introduced the extensions specifically with the following observations in mind. First, there is a need for executing tasks (e.g., payload scanning) that existing packet languages like BPF [18] cannot perform. Second, special purpose devices such as network processors or FPGAs can be quite complex and thus are not easy to program directly. Third, we should facilitate on-demand extensions, for instance through hardware assisted functions. Finally, security issues such as user authorisation and resource

constraints should be handled effectively. The previously introduced version of the FPL language [77] addressed many of these concerns. However, it lacked features fundamental to reconfigurable hardware processing like resource partition and parallel processing.

We will introduce the improved language design with an example. First, a simple program requiring a high amount of processing power is introduced in Figure 4.10. Then, the same example is discussed through multiple 'mapping' cases by using the FPL language extensions in Figures 4.11, 4.12.

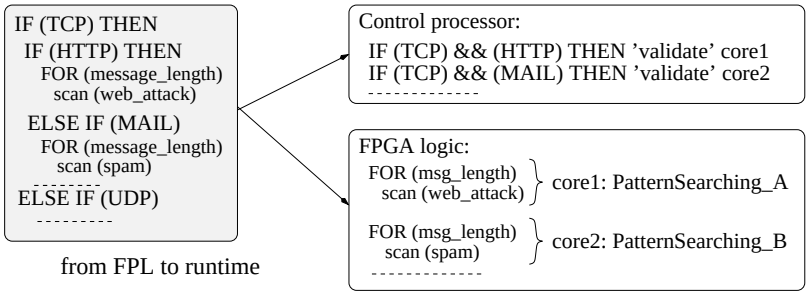


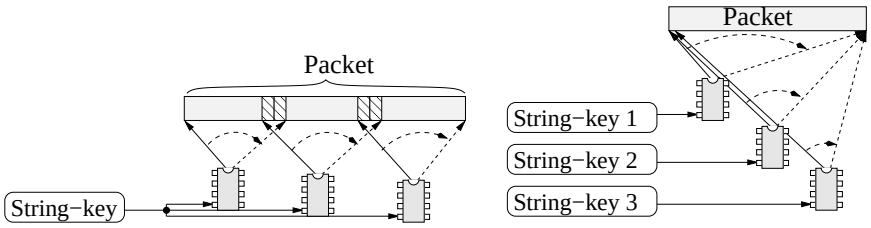
Figure 4.10: Packet processing example.

As Figure 4.10 shows, the FPL-compiler translates the program into multiple output objects, one for a control processor (ASIC embedded into the FPGA) and a second one for the FPGA reconfigurable hardware (logic that contains multiple cores). The FPGA cores consist of specific heavy computation algorithm implementations (e.g., pattern searching) that are interconnected in such way as to achieve a fast processing path as we show later in this section. Besides the parallelism built into the logic, we note that the task on the embedded control processor runs itself in parallel with the FPGA logic. The control code is mostly composed of nested IF statements used for result validation and, therefore, the processing speed of the control processor is high enough to keep up with the high speed FPGA data processing.

Note that the requirement to perform complex packet processing at gigabit line rates means that a task has a very limited time budget to process each incoming packet. When a task requires a large amount of *per-packet* processing power (e.g., a full packet scan for a worm), it becomes infeasible to perform this task on a single processing unit when network speeds go up. Thus, we take a simple byte-searching algorithm as example, and map it on the hardware using various techniques for parallel processing environment. For the sake of simplicity of the explanation, we limit our granularity to three levels of parallelism.

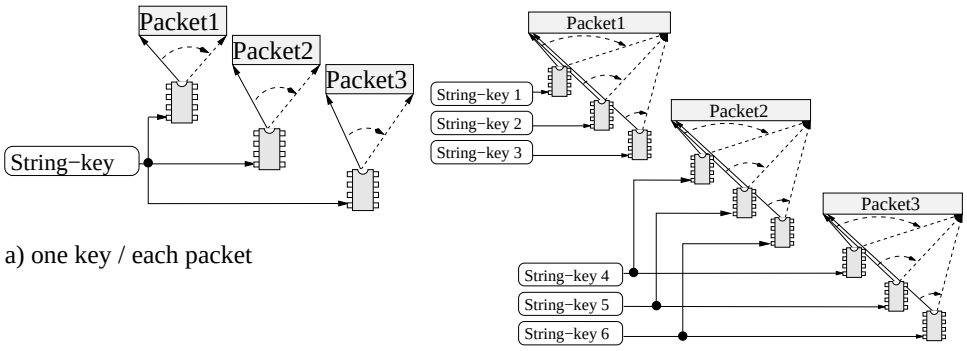
In our simple example, a processing task consists of searching through the whole packet payload data for a string (e.g., a worm signature) and it is performed by a processing unit implemented in hardware. When the task overloads the processing unit, then this task can be distributed across three hardware units in parallel, using one search key per packet, as shown in Figure 4.11.a, or multiple keys per packet (see Figure 4.11.b), or a combination of both techniques. In the first configuration, the required number of cycles is reduced with the number of hardware devices instantiated – three in our example, as the same string is searched on different parts of the packet. The second approach allows us to search in parallel three signatures on the same packet at a cycles cost of one. However, when the receiving rate is higher than the processing abilities given by 'one packet' approach, we can process

multiple packets in parallel (depth-processing), as illustrated in Figure 4.12.



a) single key / packet b) multiple keys / packet

Figure 4.11: Packet processing techniques.



a) one key / each packet

b) multiple keys / packets

Figure 4.12: Multi-Packet processing techniques.

The FPGA technology gives us enough flexibility to choose for one or a mix of the above mentioned approaches. It also provides a long-term platform life as it is easy to extend with new algorithm implementations (such as IP cores specifically designed for pattern matching, regular expressions, protocol recognition, etc.). This support may address future issues like adaptivity to new protocols (e.g., peer-to-peer). The limitation is given only by the hardware capacity and the compiler abilities to perform such complex mapping from a simple and 'natural' programming language such as FPL.

4.3.3 Using system level synthesis tool

The FPGA platform is a highly parallel structure suitable to accommodate algorithms that can exploit this parallelism. However, most commonly used imperative specification programming languages like C, Java or Matlab are hard to compile to parallel FPGA specifications because the parallelism is hard to extract. In general, specifying an application in a parallel manner is a difficult task. Therefore, we used the *Compaan* Compiler [75] that fully automates the transformation of sequential specification to an input/output equivalent parallel specification expressed in terms of a so-called Kahn Process Network (KPN). Subsequently,

the *Laura* tool [76] takes as its input this KPN specification and generates synthesizable VHDL code that targets a specific FPGA platform. The Compaan and Laura tools together realise a fully automated design flow that maps sequential algorithms onto a reconfigurable platform.

In FPL, we separate control from data intensive tasks (such as pattern matching algorithms). The latter we map on the hardware using the Compaan/Laura tool chain to implement our computationally intensive cores such as pattern matching algorithms. The former, on the other hand, can easily be mapped on a control processor of the FPGA such as PowerPC.

The FPL programming language was devised to give the FFPF platform a more expressive packet processing language than previously available. FPL conceptually uses a register-based virtual machine, and compiles to fully optimised object code. However, we had to introduce some extensions to FPL to fully exploit the FPGA features.

EXTERN(name, input, output, hw_depth) tells the compiler that the task needs the help of the specified core 'name' to process the current packet according to 'input' parameters and place the processing results in the 'output'. 'hw_depth' is an optional parameter for advanced users that want to 'force' the compiler to use a certain amount of hardware units for parallel packet processing. By default, the compiler estimates, at compile time, the *hw_depth* parameter according to the incoming traffic rate, the hardware requirements of the processing algorithm (e.g., a specific pattern matching algorithm) and its performances, and the available hardware resources.

4.4 Evaluation

In the previous sections, we described the run-time support for several hardware architectures such as commodity PCs, network processors, and FPGAs. In the following sections, we present an evaluation of the use of filters written in the FPL language on each of the supported hardware architectures.

4.4.1 FFPF on commodity PC

The FFPF architecture is arguably more complex than many of its competitors. A possible consequence of increasing expressiveness may be a decrease in performance of simple tasks. Although the FFPF architecture was introduced in Section 2.1 and an overview of the performance is described in this section, details are presented in [14]. Credits are due to Willem de Bruijn, a co-author in [14] who conducted the experiments and obtained the results presented in this section.

To verify FFPF's applicability in the general case we have directly compared it with the widely used Linux socket filter (LSF), by running identical queries through (1) libpcap with Linux' LSF back-end, and (2) libpcap based on an FFPF implementation. We realise that for various aspects of filtering faster solutions may exist, but since the number of different approaches is huge and none would be 'obvious' candidates for comparison, we limit ourselves to the most well-known competitor and compare under equivalent configurations (using the same BPF interpreter, buffer settings, etc.).

To show their relative efficiency we compare the two packet filters' CPU utilisation (sys-

tem load) using OProfile[1]. Since packet filtering takes place at various stages in kernel and userspace, a global measure such as the system load can convey overall processing costs better than individual cyclecounters. Results of sub-processes are presented, in clockcycle counts units, in Table 4.1. Both platforms have been tested with the same front-end, `tcpdump` (3.8.3). Use of the BPF interpreter was minimised as much as possible: only a return statement was executed. All tests were conducted on a 1.2 GHz Intel P3 workstation with a 64/66 PCI bus running Linux 2.6.2 with the new network API (NAPI), using FFPF with fast reader preference option and circular buffers of 1000 slots.

	task	**cycles**
1	calling a filter	71
2	single filter stage in flowgrabber	171
3	saving index in *IBuf*	154
4	storing packet in *PBuf*	7479
5	waking up user process	624
6	snort's Aho-Corasick algorithm (match)	1000
7	same but without match	9900
8	FPL filter	185
9	BPF filter	740

Table 4.1: Breakdown of various types of overhead in cycles.

Table 4.1 presents the FFPF framework overhead in the first rows (rows $1-4$), and filtering overhead in the last rows (row $5-9$). Furthermore, comparing filtering and framework overhead shows that costs due to FFPF's complexity contributes only a moderate amount to overall processing. Finally, we discuss in the next section that the IXP implementation is able to sustain full gigabit rates for the same simple FPL filter that was used in Table 4.1, while a few hundred Mbps can still be sustained for complex filters that check every byte in the packet [73]. As the FPL code on the IXP is used as pre-filtering stage, we are able to support line rates without being hampered by bottlenecks such as the PCI bus and host memory latency, which is not true for most existing approaches.

We note that the results presented in this section were obtained in 2003 and 2004. Since then, the same research group has continued the development of packet processing languages for IXPs, showing in their Ruler paper [27] that it is possible to push performance of full packet inspection on IXPs to beyond gigabit rates.

4.4.2 FFPF on NP: NIC-FIX

NIC-FIX was designed to scale with link rates, but as we do not have a 10 Gbps testbed, we evaluate the architecture with a setup using IXP1200 (details in Section 2.2.2), while monitoring a gigabit link. We deliberately used an 'obsolete' network processor such as IXP1200 which belongs to the generation of single gigabit link-rate.

[1] `http://oprofile.sourceforge.net`

```
// count number of packets in every flow,
// by keeping counters in hash table (of size 4095)
IF (PKT.IP_PROTO == PROTO_TCP) THEN
  R[0] = Hash(26, 12, 4095);    // hash over TCP flow fields)
  M[ R[0] ]++;                  // increment the pkt counter at this position
FI;
```

Listing 4.2: (B) - count TCP flow activity.

On-board processing

An important constraint for monitors is the cycle budget. At 1 Gbps and 100 byte packets, the budget for four threads processing four different packets is almost 4000 cycles. As an indication of what this means, Table 4.2 shows the overhead of some operations. Note that these results include all boilerplate (e.g., transfers from memory into read registers and masking). Comparing the operations cycle costs against the cycle budget, we can say that a filter may use, for instance, one hash over the flow fields and 31 words read out of a received packet.

Description	Value
R[0] = HASH(26, 12, 255)	200 cycles
R[0] = PKT.B[0]	110 cycles
R[0] = PKT.W[0]	120 cycles

Table 4.2: Approximate overhead of some operators.

To evaluate NIC-FIX, we execute the three filters shown in Listings 4.1, 4.2, 4.3 on various packet sizes and measure throughput. Only A is a 'traditional' filter that checks some header fields and then classifies the packet. The other two gather information about traffic, either about the activity in every flow (assuming the hash is unique), or about the occurrence of a specific byte. Note that the hash function used in B utilises dedicated hardware support. We implemented three variations of filter C. In $C1$, the loop does not iterate over the full packet, just over 35 bytes (creating constant overhead). In $C2$, we iterate over the full size, but each iteration reads a new quadword (eight bytes) rather than a byte. $C3$ is filter C in Listing 4.3 without modifications. The results are shown in Figure 4.13.

```
IF (PKT.IP_PROTO == PROTO_UDP && PKT.IP_DEST == X && PKT.UDP_DPORT == Y)
    THEN RETURN 1;
    ELSE RETURN 0;
FI;
```

Listing 4.1: (A) - filter packets.

Figure 4.13 shows that above a packet size of 500 bytes, NIC-FIX can process packets at line rate for A, B and $C1$. This means that if the traffic consisted of packets that match filter A, the prefiltering in NIC-FIX ensures applications like `tcpdump` would also handle link rate.

For smaller packets, filters $C1 - 3$ are not able to process the packets within the cycle budget. Up to roughly 165.000 packets per second $C1$ still achieves throughput of well above

```
IF (PKT.IP_PROTO == PROTO_UDP ) THEN
  R[0] = PKT.IP_TOTAL_LEN;                  // saved pkt size in register
  FOR (R[1] = 0; R[1] < R[0]; R[1]++)
     IF (PKT.B[ R[1] ] == 65) THEN          // look for char 'A'
          R[2]++;                           // increment counter in register
     FI;
  ROF;
  M[0] = R[2];                              // save to shared memory
FI;
```

Listing 4.3: (C) - count all occurrences of a character in a UDP packet.

900 Mbps. Beyond that, the constant overhead cannot be sustained. $C2$ and $C3$ require more cycles for large packets and, hence, level off sooner. This suggests that simple pre-filters that do not access every byte in the payload are to be preferred. This is fine, as the system was intended precisely for that purpose.

Just as for the C filters, throughput also drops for the simple filters A and B when processing smaller packets. However, these drops occur for a different reason, namely because we used a commodity PC with a gigabit PCI card for traffic generator and hence, the PCI bus limits the throughput (especially for small packets when many interrupts need to be handled).

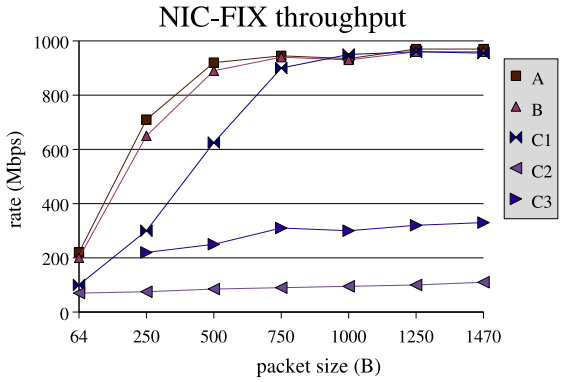


Figure 4.13: Throughput for different NIC-FIX filters.

4.4.3 FFPF on FPGA: NIC-FLEX

FPL is a source-to-source compiler and generates C target code that can be further handled by any C compiler. Programs can therefore benefit from the advanced optimisers in the Intel μEngine C compiler for IXP devices, `gcc` for commodity PCs, and `gcc` for the PowerPC control processor on Xilinx's FPGAs. As a result, the object code will be heavily optimised even though we did not write an optimiser ourselves.

Moreover, the FPL compiler uses a heuristic evaluation of the hardware instances needed to reach the system goal, where the goal may be to 'perform a certain algorithm for packets at a line rate of 1Gbps'. The evaluation is based on the workload given by one hardware instance to perform the user's program and a critical point where the performances drop because of some heavy computation like signature length, or packet size. For example, in Figure 4.12.b, assuming that the user's program performs checking of six signatures, but three of them are known as much longer than the others, the compiler duplicates the hardware units, accordingly, in order to achieve a well balanced workload of the whole system.

Control processor and FPGA cores

Embedded in modern FPGAs are one or more hard core control processors (e.g., PowerPC or ARM), on which we may map the control part of our algorithms. The data intensive tasks, on the other hand, are mapped directly in hardware (as IP cores) using the Compaan/Laura tool chain. The IP cores communicate with the control processor using a set of registers. For instance, these registers are used to set some run-time parameters (e.g., the packet length, or the searched key-strings). In other words, the FPL compiler extracts the affine loops from FPL into cores with the help of Compaan/Laura tool and uses the rest of the FPL code to control the former cores.

To study the feasibility of using the Compaan/Laura tool chain in the Networking world we compiled in hardware a search algorithm. The Matlab program for this algorithm is shown in Listing 4.4. At line 8, the bytes of the packet (pkt) are compared with the content of a signature string (sig). If the signature is present in the packet, then the value of the c variable is equal to the length of the searched string.

```
1  for i = 7: 1: PackSize,
2    c = 0;
3    for j = 1: 1: StringLength,
4      if sig(j) = pkt(i),
5         c = c + 1;
6      end
7    end
8    if c = StringLength,
9      print "Found!"
10   end
11 end
```

Listing 4.4: Simple Search Algorithm.

The program shown in Listing 4.4 has been rewritten to match the requirements of the Compaan/Laura tool chain. Additionally, we instructed our tool to generate a design that compares eight characters in parallel. The hardware network of processors is depicted in Figure 4.14. Each bubble represents a hardware processor and each arch a communication channel between two processors. The *ReadPacket* processor feeds our network with packet bytes from a MAC network interface. The *Search* processor implements the character-wise searching, the result of 'a search' is evaluated by the *Eval* processor, which is also our write interface toward external devices.

Table 4.3 gives the hardware results of the FPGA implementation of the algorithm given in Listing 4.4. The experiment has been conducted using Synplify and ISE Xilinx 6.2 for the

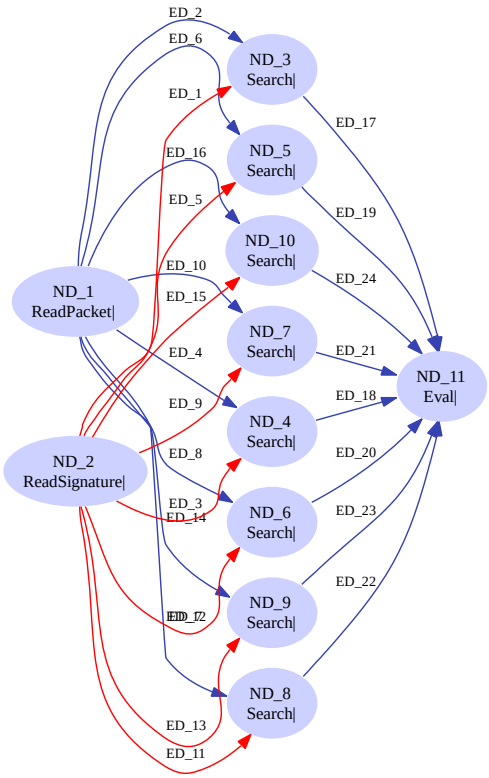


Figure 4.14: Processor Network of the simple algorithm.

Virtex II-6000 platform. The hardware is capable of doing an eight character string search in a variable packet size. The required number of cycles for a variable packet size and eight characters search string is $cycles = 13 + PacketSize$.

Packet Length	String Length	Clocks/Workload	Slices	Frequency (MHz)
64	8	77	2035	101

Table 4.3: Experimental results.

In our example, the length of the search string is fixed to eight characters. However, the string size can be changed at compile time while its content may be changed at runtime. Although the string search algorithm presented above works on only one string and was shown for simplicity, for searching of multiple strings other algorithms (e.g., Aho-Corasick) may be implemented.

NIC-FLEX evaluation

Given the pattern matching algorithm result (see Table 4.3) for one search-key per packet, we extrapolate to other case studies as already illustrated in Figure 4.12. In Figure 4.15 is shown how the performance of one key per packet approach (1key/1hw) scales up by increasing the use of hardware units (1key/3hw) in parallel. Note that unlike general purpose processors, FPGAs allow us to extrapolate performance, due to the predictable nature of the hardware: we know exactly what will happen in every clock cycle.

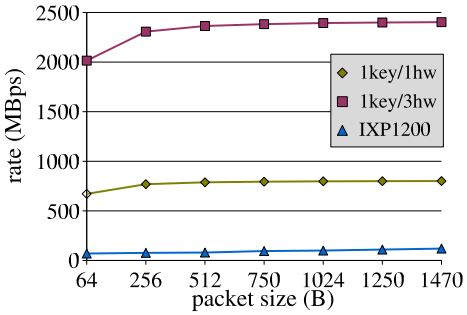


Figure 4.15: FPGA vs. NP processing results.

The processing result of a full packet payload pattern search filter performed by a 1Gbps generation network processor (Intel IXP1200) is also shown in Figure 4.15. Therefore, making a comparison between an FPGA implementation and a network processor implementation, it can be seen that a complex filter (such as a pattern searching algorithm) performed by a NP is surpassed by even a single IP core implementation.

Note that the relatively small amount of hardware resources used for this implementation (ca. 6 % for a Virtex II-6000) allows us to map more than one search engine into a FPGA platform.

4.5 Summary

In this chapter we saw what techniques we use to provide traffic processing at high speeds by using tightly coupled parallelism through multi-cores offered by commodity PCs, network processors, and FPGA chips. However, for higher speeds, we propose in the next chapter to increase the parallelism that can be exploited beyond a tightly coupled system by means of a distributed processing environment.

Chapter 5

Distributed Packet Processing in Multi-node NET-FFPF

5.1 Introduction

We have shown that parallelism can be exploited to deal with processing at high speeds. A network processor (NP), for example, is a device specifically designed for packet processing at high speeds by sharing the workload between a number of independent RISC processors. However, for very demanding applications (e.g., payload scanning for worm signatures, or custom video streams processing) at future link rates more power is needed than any one processor can offer. For reasons of cost-efficiency it is unpractical to develop NPs that can cope with backbone link rates for such applications. Building a NP that supports only a few applications at high speeds is not cost-efficient because such a product has a small market. An attractive alternative is to increase scalability by exploiting parallelism at a coarser granularity.

We have previously introduced an efficient monitoring framework, Fairly Fast Packet Filters (FFPF) (in Chapter 2), that can reach high speeds by pushing as much of the workload as possible to the lowest levels of the processing stack. The *NIC-FIX* architecture (in Chapter 3) showed how this monitoring framework could be extended all the way down to the network card. To support such an extensible programmable environment, we introduced the special purpose FPL language.

In this chapter, we exploit packet processing parallelism at the level of individual processing units (NPs or commodity PCs) to build a heterogeneous distributed monitoring architecture: NET-FFPF. Incoming traffic is divided into multiple streams, each of which is forwarded to a different processing node (Figure 5.1). Simple processing occurs at the higher levels (the root nodes), while increasingly more complex tasks take place in the lower levels where more cycles are available per packet. The main contribution of this chapter consists of an extension to the language introduced previously with a feature of distribution of complex packet processing tasks. Also, with NET-FFPF we extend the NIC-FIX architecture upwards, with packet transmission support, to create a distributed filtering platform. Experiments show

NET-FFPF to be able to handle complex tasks at gigabit line-rates.

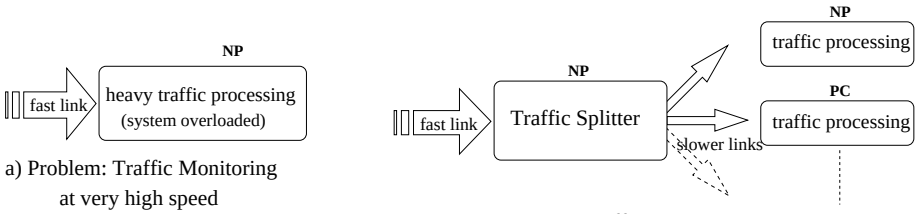


Figure 5.1: Moving to distributed traffic monitoring.

NET-FFPF builds on the idea of traffic splitting that was advocated by Charitakis *et al.* in [8] and Kruegel *et al.* in [7] for intrusion detection. However, we use it to provide a generic high-speed packet processing environment. Charitakis *et al.* focus on packet *header* processing and automatically generate the appropriate code for the splitter (implemented on a network processor) from high-level snort rules. They show that traffic splitting improves the packet processing performance even if the splitter and all processing nodes reside on the same host. The traffic slicer in [7] employs a two-stage approach for intrusion detection where rules for traffic splitting are formed by modelling the attacks.

The NET-FFPF implementation resembles the slicer in that it also mangles Ethernet frames to split the traffic. At a more fundamental level, however, NET-FFPF differs from both of the above approaches in that it allows for processing hierarchies that are arbitrarily deep and heterogeneous, whereby each level performs a part of the total computation. Moreover, NET-FFPF offers explicit support for such processing at the language level. By applying the splitter concept in a distributed fashion NET-FFPF can facilitate such diverse tasks as load balancing, traffic monitoring, firewalling and intrusion detection in a scalable manner, e.g., in enterprise gateways.

5.2 Architecture

In this section we describe the extensions to the FPL language and compiler with specific constructs so as to provide code splitting over a distributed processing environment.

5.2.1 High-level overview

At present, high speed network packet processing solutions need to be based on special purpose hardware such as dedicated ASIC boards or network processors (see Figure 5.1a), although faster than commodity hardware, even solutions based on these platforms are not sustainable in the long run because of a widening gap between growth-rates in networking (link speed, usage patterns) and computing (cpu, main memory and bus speed), as we have argued in Section 1.4.

To counter this scalability trend we propose the solution shown in Figure 5.1b, which consists of splitting the incoming traffic into multiple sub-streams, and then processing these individually. Processing nodes are organised in a tree-like structure, as shown in Figure 5.2.

By distributing these nodes over a number of possibly simple hardware devices, a flexible, scalable and cost-effective network monitoring platform can be built.

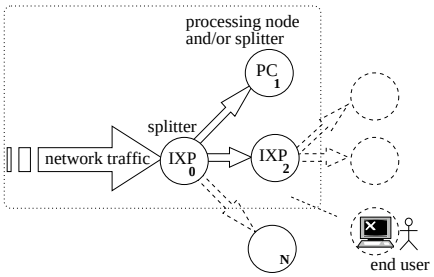


Figure 5.2: Distributed system overview.

Each node in the system performs a limited amount of packet processing (e.g., filtering, sampling) and may split its incoming stream according to some arbitrary criteria into multiple output streams that are sent to different nodes at lower levels. For example, all TCP traffic is sent to node N_1, all UDP traffic to node N_2. As the traffic arrives at node N_0 at the full link rate, there will be no time for complex packet processing on this node, due to the limited cycle budget. Therefore, at this node we perform only a simple classification of packets into substreams. Each substream's packets are forwarded to a dedicated node at the next level in the tree. In general, we do not restrict classification to the number of processing nodes in the next level. In other words, it may happen that packets of class X and packets of class Y are sent to the same node at the next level. It also may be necessary to multicast packets to a number of nodes. For instance, if all TCP traffic is sent to N_1, the traffic will overlap with both a stream of all HTTP traffic sent to N_2 and a stream of SMTP traffic sent to N_3. In such a scenario, packet duplication is needed.

The demultiplexing process continues at the next levels. However, the lower we get in the hierarchy, the fewer packets we need to process. Therefore, more complex tasks may be executed here. For instance, we may want to perform signature matching or packet mangling and checksum recalculation. In principle, all non-leaf nodes function as splitters in their own rights, distributing their incoming traffic over a number of next level nodes.

Note that a tree-like hierarchy is a natural fit for our distributed processing, but other organisations are possible. For instance, one may need to re-process a stream and hence, the traffic must travel forth and back over the tree. We have not explored such configurations in this thesis.

5.2.2 Distributed Abstract Processing Tree

The introduced networked processing system can be expressed in a distributed abstract processing tree (D-APT) as depicted in Figure 5.3. For an easier understanding of the D-APT functionality, we use the following notations throughout the text. A D-APT is a tree composed of individual APTs, each of which runs on its own dedicated hardware device. An APT is built up of multiple processing elements (e.g., packet filters) and may be interconnected to other APTs through so-called *in-nodes* and *out-nodes*. *In-nodes* and *out-nodes* are

processing elements that perform, in addition to the user filters, receiving and transmission tasks, respectively. For example, $N_{0.3}$, $N_{0.5}$ are out-nodes, while $N_{1.1}$, $N_{2.1}$ are in-nodes.

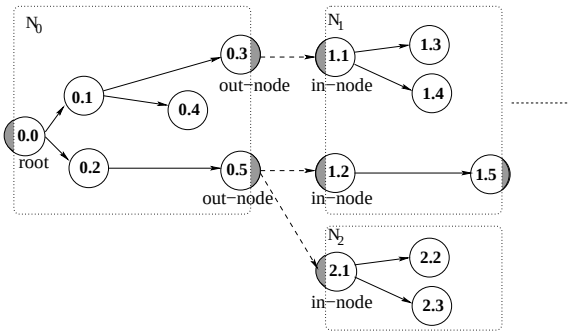


Figure 5.3: Distributed Abstract Processing Tree.

By ordering the processing nodes, APTs also describe the traffic streams that flow between them. The incoming stream is decomposed into multiple substreams. Simple processing is performed at the higher levels, on the left, while more complex processing happens in the lower levels, on the right (see Figure 5.3). The idea is to balance the amount of traffic and processing for each node.

As an APT represents traffic splitting as well as processing, a stringent requirement is that processing at any level N continues exactly where it left off at level $N - 1$. We can achieve this by explicitly identifying the break-off point, as we will see later.

We note that the performance of the whole distributed monitoring system is determined by the total number of the processing nodes, the processing power of each node, as well as the distribution of tasks and traffic over the nodes.

Besides our architecture advantages, we also mention a drawback of using of a 'distributed model' for traffic processing: mapping of existing applications (e.g., intrusion detection systems) often requires a full redesign in order to parallelise sequential application into multiple and parallel tasks.

5.2.3 Extensions to the FPL language

With FPL, we adopted a language-based approach, following our earlier experiences in this field. Recall that we designed FPL specifically with the following observations in mind. First, there is a need for executing tasks (e.g., payload scanning) that existing packet languages like BPF [18], Snort [46] or Windmill [23] cannot perform. Second, special purpose devices such as network processors can be quite complex and thus are not easy to program directly. Third, we should facilitate on-demand extensions, for instance through hardware assisted functions. Finally, security issues such as user authorisation and resource constraints should be handled effectively. The previous version of the FPL language, presented in Chapter 3, addressed many of these concerns. However, it lacked features fundamental to distributed processing like packet mangling and retransmission. In other words, in addition to 'traditional' traffic monitoring (e.g., filtering, counting), a distributed traffic monitoring system needs to be able

to send possibly modified packets to remote hosts for further processing. Moreover, the architecture needs to ensure that processing at a remote host continues exactly where the previous node left off (we refer to this as the break-off point), so we need to add this mechanism to our framework as well.

We will introduce the language design with an example. First, a program written for a single machine (N_0) is shown in Figure 5.4. Then, the same example is 'mapped' onto a distributed abstract processing tree of tree nodes (N_0, N_1, N_2) by using the new language extensions in Figure 5.5.

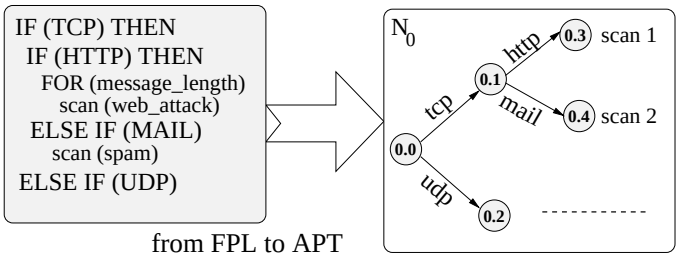


Figure 5.4: Traffic monitoring program mapped onto APT.

Figure 5.4 shows how the full stream is split at the root node $N_{0.0}$ into two sub-streams: `TCP` and `UDP`. Then, the `TCP` sub-stream is again split into two smaller sub-streams, `http` and `mail`, by the intermediate node $N_{0.1}$. The actual processing tasks of scanning for web attacks and spam, are denoted by $N_{0.3}$ and $N_{0.4}$, respectively. On the right side of the figure we see how the data flows in a simple data flow graph, where edges represent data streams and vertices represent processing steps.

Let us now assume that the amount of traffic of HTTP and email traffic is too high to allow both scanning tasks to be collocated at the same node. Really, we want to execute the same work-flow, but run on multiple nodes. For this purpose, we introduce an explicit *SPLIT* language construct that splits incoming traffic and forks of the computation into a SPLIT block to a node in the next processing level.

Figure 5.5 now gives the same example, written using the `SPLIT` extension for a distributed processing environment, taking the hardware depicted in Figure 5.2 as environment. For the sake of simplicity of explanation, we limit our tree to two levels. The program is mapped into the D-APT shown in Figure 5.5 by taking into account both the user request (our program) and a description of the hardware configuration. As Figure 5.6 shows, the FPL-compiler takes both files, the FPL program and the hardware description, performs mapping of the program onto the specific distributed system, and as result, it generates a code object for each processing node. The object files are transferred and loaded remotely with the help of the FFPF 'management' toolkit.

One of the main differences between running on a simple shared memory system and our distributed processing environment is that sharing state is no longer feasible because synchronising data across nodes is too costly relative to the high-speed packet processing task itself. However, we will show how we are able to distribute a small amount of state to downstream processing nodes in Section 5.3.

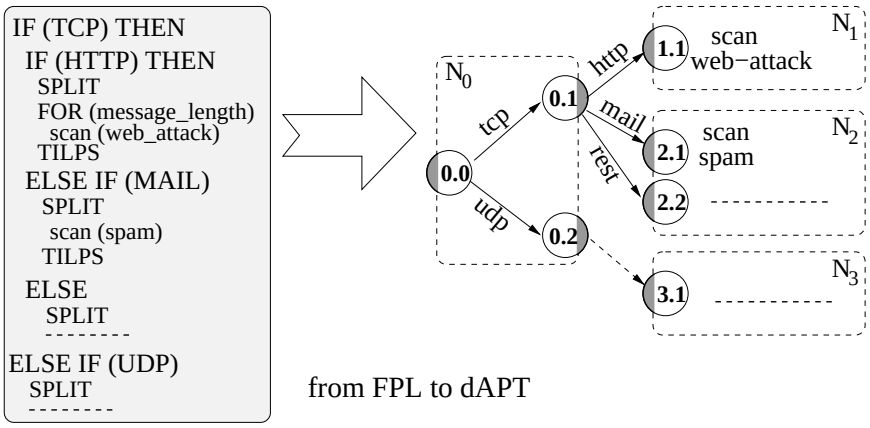


Figure 5.5: Mapping an APT to a D-APT using FPL's SPLIT command.

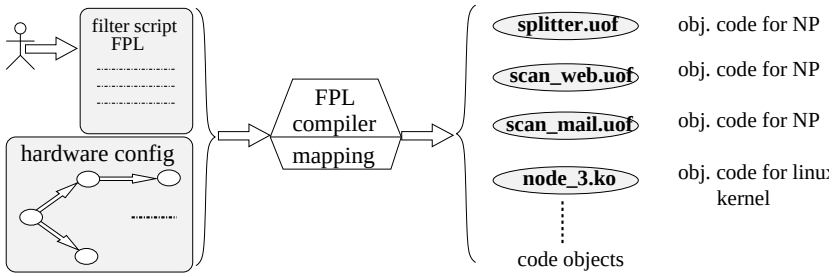


Figure 5.6: User compiles an FPL filter expression for a distributed system.

5.3 Implementation

NET-FFPF builds on FFPF, the monitoring framework designed for efficient packet processing on commodity hardware, such as PCs. As we have seen, FFPF offers support for commonly used packet filtering tools (e.g., tcpdump, snort, libpcap) and languages (like BPF), as well as for special purpose hardware like network processors (NPs) and hardware reconfigurable (FPGAs). However, it is a single-device solution. NET-FFPF extends it with distributed processing capabilities through language constructs. Currently, we have support for the following target platforms: (1) IXP1200 network processors, (2) IXP2400 network processors, (3) IXP2850 network processors, Virtex II-Pro FPGA, and (4) off-the-shelf PCs running Linux. The execution environment and the FPL language are presented in detail in the next section.

As described in Section 5.2, the architecture of distributed traffic processing uses two concepts, as follows: (1) split the application into a distributed system by means of multiple tasks running on different processing nodes, and (2) interconnect the tasks into a processing hierarchy by means of receiving and transmitting traffic between processing nodes. Therefore, we need to extend our FPL language and the processing environment with two features: splitting of the application into multiple tasks, and transmission of traffic between multiple

nodes.

The FPL extensions syntax is summarised in Table 5.1 and described afterwards. For the full language syntax see the FPL constructs in Section 3.1.2.

transmit to	TX(table_type, table_index) or TX(table_type, table_index, origin)
split the code	SPLIT; stmts; TILPS or SPLIT(node_index); stmts; TILPS
shared states	MEM namespace[size] or MEM[size]

Table 5.1: FPL language constructs (extensions to the FPL).

In our implementation, we assume Ethernet links and focus on the traffic transmission at the link layer due to the following reasons: (1) our goal is to process traffic at multi-gigabit link-rates and we assume that the entire distributed system is located on the same network, and (2) we avoid having to re-compute a checksum on every packet by software (e.g., IP-checksum) because the Ethernet cyclic redundancy check (CRC) is automatically performed by the hardware at each egress point. In the future we may also consider higher layer protocols such as IP.

TX() construct.

The purpose of this construct is to schedule the current packet for transmission. Amongst the simplest physical interconnection, such as a direct fiber between two nodes, we can also have gigabit switches between processing nodes. Therefore, we need to tell the switches what is the destination of a particular packet. In this respect, the TX() operation involves overwriting the Ethernet destination address (`ETH_DEST`) of a packet with an entry from a programmable MAC_table (`TX_MAC`). By doing so, we let the switches to deliver the packets according to the 'mangled' `ETH_DEST` field.

The use of the first type of transmission operation, `TX(table_type,table_index)`, is illustrated in Listing 5.1.

```
1  TX_MAC[3] = {00:00:E2:8D:6C:F9, 00:02:03:04:05:03, 00:02:B3:50:1D:7A};
2  // extracted by the compiler from the configuration file
3  IF (PKT.IP_PROTO == PROTO_TCP) // if the packet is TCP
4    THEN TX (Mac, 2);            // schedule it to be forwarded to the 3rd
5    ELSE TX (Mac, 1);            // or 2nd MAC address from the TX_MAC table
6  FI;
```

Listing 5.1: A simple usage of TX(table_type, table_index).

In this example, the first TX parameter selects a table type (MAC or another field, such as `IP_DEST`, in a future implementation) and the second parameter points the index in the table.

Note that by inserting multiple TX() calls into the same program we can easily implement packet replication and load-balancing, as shown by the example in the Listing 5.2 and briefly described as follows. When the received packet is TCP then it is transmitted to two hosts

the MAC addresses of which are pointed to by the first and the third entry in the 'TX_MAC' table. If the received packet is not TCP, then a persistent state counter is updated, 'M[0]', based on which we have implemented a primitive form of load balancing. For example, the first non-TCP packet goes to the host pointed to by the first MAC entry and the second non-TCP packet goes to the host pointed to by the second entry in the 'TX_MAC' table.

We also note in this example the way the runtime system avoids multiple packet copies, as generally described in Section 4.2.1 and 4.2.2. When the program decides to transmit a packet then the runtime will put the packet's reference (an index only) into the transmission queue. Next, the transmission task takes each enqueued packet's reference, updates some specific fields in the packet (e.g.,ETH_DEST), and then transmits the packet. However, a packet may be replicated by placing the same packet's reference for multiple times into the transmission queue. The transmitter would then transmit all packet's references regardless they point to the same packet or not.

```
1  TX_MAC[3] = {00:00:E2:8D:6C:F9, 00:02:03:04:05:03, 00:02:B3:50:1D:7A};
2  MEM[1];
3  IF (PKT.IP_PROTO == PROTO_TCP)    // if packet is TCP
4    THEN
5      TX (Mac, 0);          // replicate the packet to the 1st MAC_ADDR
6      TX (Mac, 2);          // replicate the packet to the 3rd MAC_ADDR
7    ELSE
8      M[0]++;
9      IF (M[0]%2)           // load balance it over the 1st and 2nd MAC_ADDR
10       THEN TX (Mac, 0);
11       ELSE TX (Mac, 1);
12     FI;
13 FI;
```

Listing 5.2: Packet replication and load balance using TX().

We also provide a second TX construct when using the optional parameter $origin$. The $origin$ is a unique identifier in an FPL program that needs to be mapped on a distributed environment. The $origin$ helps to locate the current processing point being interrupted by a TX (the break-off point) on the next processing environment node. In other words, the $origin$ provides the current node with the out-node of the previous processing level.

Although this TX construct is available to users through a language construct, it is really meant to be used by the FPL-compiler when it detects SPLIT constructs, as we will see later.

SPLIT() construct.

To explain SPLIT we will step through the example in Figure 5.5. When trying to match the given FPL filter to a distributed system, the compiler detects the SPLIT construct. SPLIT tells the compiler that the code following and bounded by the corresponding TILPS construct can be split off from the main program. The example script is split into subscripts as follows: one script is run on the splitter node N_0, and four more on each processing node N_1, N_2, and N_3, as shown in Figure 5.7. In other words, we have four FPL source files that are each compiled into object code by a native compiler (e.g., MicroC compiler for a network processor) and then integrated into the distributed processing framework with some boiler plate, as we will see later.

The current implementation is based on Ethernet header mangling, implicitly driven by TX constructs. The destination address of an Ethernet packet (`ETH_DEST_ADDR`) is overwritten to the device containing the next node in the D-APT. Recall that one of the NET-FFPF requirements is that processing at level N continues exactly where it had broken off at level $N-1$. The way we implemented this in the Ethernet implementation is by using the Ethernet source address to identify the out-node at level $N-1$. However, there is no need to use all six bytes of the source address for this purpose. For this reason we split the source address (`ETH_SRC_ADDR`) into two identifiers: *processing state* (four bytes) and *origin indicator* (two bytes).

The origin indicator specifies the out-node of the previous level (allowing for 64K outpoints), while the 32 bit state field can be used in an application-specific way. In practice, we use it to allow nodes at level $N-1$ to pass a limited amount of information to the next processing node. For instance, if a node at level N has generated a semi-unique identifier for the packet (e.g., as a 32bit hash value), we can pass this value to all nodes down in the processing hierarchy, so they will not have to repeat the expensive hash. As shown in Figure 5.7, we can now efficiently continue the computation at level N, by using a `switch` statement on the origin indicator to jump to the appropriate point to resume. Observe that a `switch` statement is only needed if more than one outnode is connected to this in-node. Also observe that although we have not implemented this, it may be possible to generalise NET-FFPF beyond subnets by basing the splitter functionality on a tunnelled approach or by overwriting IP header fields, instead.

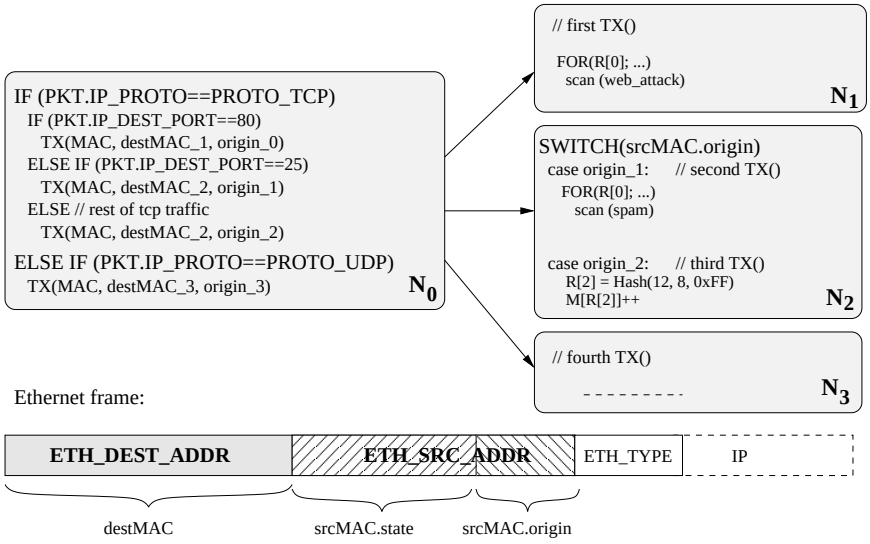


Figure 5.7: SPLIT in detail.

The compiled example program is executed as follows. The code at the root node (N_0) deals only with selective forwarding. Any packet that matches one of the IF statements has its Ethernet address fields modified and is then forwarded to the next-level nodes. The other nodes will only have to process these forwarded packets. Node N_2 for instance, receives

```
1  MEM M1[100]; // reserve 100 Bytes for use in node 1 (namespace M1)
2  MEM M2[150]; // reserve 150 Bytes for use in node 2 (namespace M2)
3  MEM[2];      // reserve 2 Bytes for a global "one-way" shared state
4  IF(PKT.IP_PROTO == PROTO_UDP) THEN
5    SPLIT;
6    IF(PKT.IP_DEST == 100 && PKT.UDP_DPORT == 3100) THEN
7      M1[0]++; // local counter
8      M[0]++;  // global counter
9    FI;
10   TILPS;
11   IF(PKT.UDP_DPORT == 1434) THEN
12     SPLIT;
13     FOR(R[0]=7;R[0]<PKT.IP_TOTAL_LEN;R[0]++)
14       IF(PKT.DW[R[0]] == 0x65676869) THEN M2[1]++ FI
15     ROF;
16     TILPS;
17   FI;
18 FI;
```

Listing 5.3: State sharing between nodes using namespaces.

two classes of packets forwarded from node N_0. As illustrated in Figure 5.7, in each of the *SWITCH* blocks and in the Ethernet frame, the classes are identified by the `origin` indicator embedded in the `ETH_SRC_ADDR` field.

Note that by passing the optional argument 'node_index' to `SPLIT`, a user can force packets to be forwarded to a specific node, as in the example that we will show in Section 5.4. This can be useful when a node has special abilities well-suited to a given task, e.g., hardware-accelerated hashing.

MEM construct.

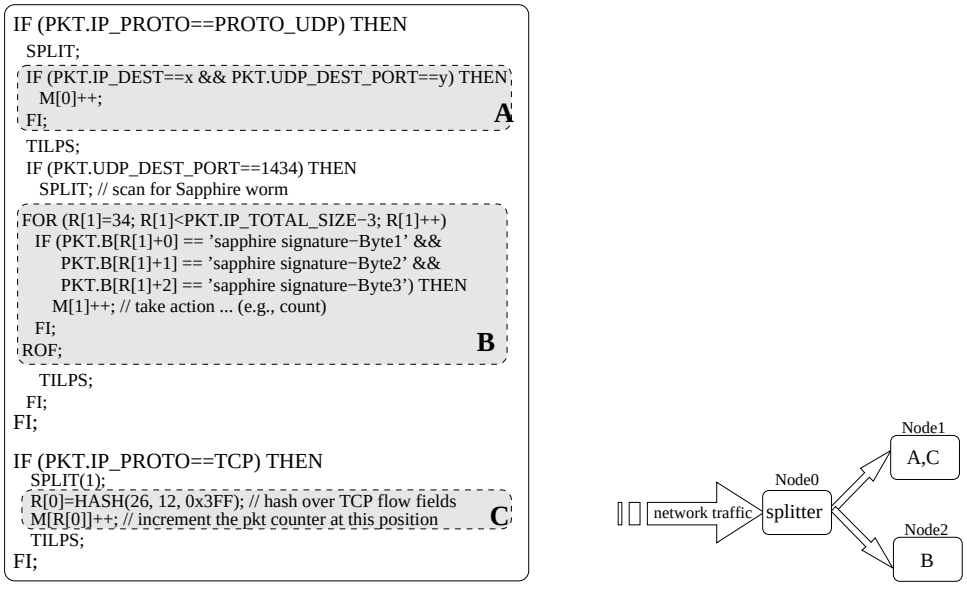
This construct instructs the compiler about the user's wishes regarding state sharing. As shown in Listing 5.3, the MEM's parameter identifies a certain namespace used within a node from the distributed environment and identified by the SPLIT construct.

The namespace usage is introduced in order to help the user to use a common shared state over the entire processing hierarchy and the individual sharing states within one processing node. The common state is limited in size (4 Bytes in our implementation), while the individual state is theoretically unlimited, but limited by the hardware. Moreover, in our processing hierarchy model, the traffic data flows from root down in the hierarchy, the shared state is synchronised only in one way. In order to implement the programming restrictions for namespaces, the compiler allows only a 4 bytes limited default namespace ($MEM[]$ without namespace identifier), and one custom namespace usage within a node identified by SPLIT/TILPS code block ($MEMcustomName[]$).

Although a shared state between nodes is very limited in size, in our Ethernet implementation it is possible to transfer a state of maximum 4 Bytes from the root node down in the processing hierarchy. The MEM[] construct implements the global namespace across the distributed nodes.

5.4 Evaluation

To evaluate NET-FFPF, we take a complex FPL filter, execute it on various packet sizes and measure the throughput. For instance, the filter shown in Figure 5.8.a is composed of three tasks: a simple packet counter (task A), a heavy processing task such as a packet scanner (tasks B), and a packet counter of flows (task C). This filter is mapped onto a distributed system composed of three nodes, as shown in Figure 5.8.b. We map the tasks in such a way that tasks A and C run on one node ($Node_1$) and task B runs on a second node ($Node_2$). In addition, there is a root node ($Node_0$) that splits the incoming traffic into two flows. The compilation results are the code objects of the splitter, and those of the two sub-filters: tasks A+C, task B.



(a) Filter (b) Distributed system

Figure 5.8: Mapping a complex filter on a distributed system.

We note that only A is a traditional per-packet counter. The other two gather per-flow information. As the hash function used in C utilises dedicated hardware support, we push (by providing the parameter `node_index`) the C filter onto the same hardware node as A filter: $Node_1$. In B, a loop iterates over the whole `UDP` packet payload. While the filters A and C are easily performed on an NP even at high speed (1Gbps), the B filter incurs so much processing that even an IXP1200 network processor cannot handle its stream at such a speed (see Figure 5.9). As the processing results show, using the above mentioned mapping, we can process the full packet data up to only ca. 100 Mbps.

Supposing we need to process all filters at gigabit speed, we would let the B filter consequently split to another node and we can successfully process all the packets for a particular UDP port (assuming the packets related to a specific worm are within 100 Mbps bandwidth). If more bandwidth is needed, then more processing nodes have to be involved.

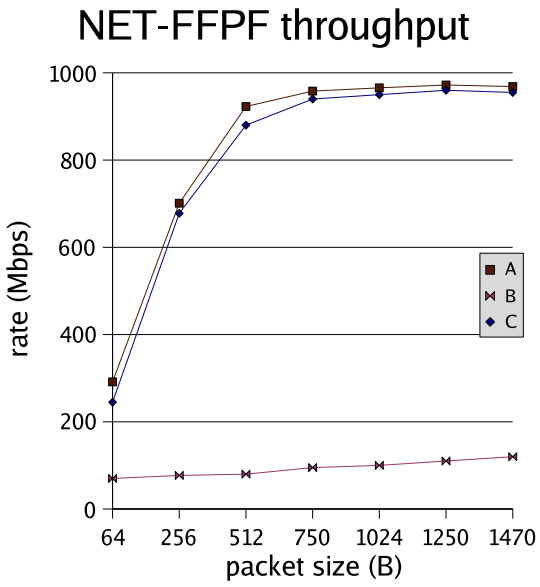


Figure 5.9: Benchmark result.

```
1  IF (PKT.IP_PROTO==TCP && PKT.IP_DEST_PORT==80) THEN
2   R[0] = hash(flowfields, 3);
3   SPLIT(R[0]);                        // thus, the main stream is equally
4    <scan for web-traffic worms>       // distributed across of 3 processing nodes
```

Listing 5.4: An example of load balancing for web-server.

As illustrated in Figure 5.9, just as for the B filter, the throughput also drops for the simple filters A and C when processing smaller packets. However, these drops occur for a different reason, namely because we used a commodity PC with a gigabit PCI card for traffic generator and hence, the PCI bus limits the throughput (especially for small packets when many interrupts need to be handled).

Demonstrating the simplicity a concise special purpose language like FPL brings, a naive load balancing program for a web-server is shown in Listing 5.4. A hash over the flow fields of each incoming packet determines to which node the packet is forwarded. In a handful lines of code the web traffic is split into three equal substreams. Doing so, the TCP flows are kept together to the benefit of the processing applications on the distributed nodes and to an easier retrieval and merging of the processing results from the distributed nodes.

5.5 Summary

Summarising, this chapter presented the NET-FFPF distributed network processing environment and its extensions to the FPL programming language, which enable users to process network traffic at high speeds by distributing tasks over a network of commodity and/or special purpose devices such as PCs and network processors. A task is distributed by constructing a processing tree that executes simple tasks such as splitting traffic near the root of the tree while executing more demanding tasks at the leaves. Explicit language support in FPL enables us to efficiently map a program to such a tree. The experimental results show that even on hardware that is now considered obsolete, we can process packets in computationally expensive applications at rates ranging from 100 Mbps to 1 Gbps.

The next chapter extends NET-FFPF with a control environment that takes care of code compiling, object code loading, and program instantiation. These control actions perform automatically as a response to changes in the system's environment like the increase of specific traffic (e.g., tcp because of a malicious worm) or availability of new hardware in the system such as a system upgrade.

Chapter 6

Towards Control for Distributed Traffic Processing: Conductor

6.1 Introduction

The increasing complexity in traffic processing leads to a control problem when using large-scale, high-throughput distributed systems in dynamic environments. Specifically, running the applications onto a distributed environment challenged by dynamic and unpredictable events (e.g., traffic peaks, hardware/software failures) gives the following control problem: "keep the system stable for any environment change".

In previous chapters we have seen that although there are specifically designed parallel architectures for packet processing such as network processors (NPs) [48, 56, 78]. In practice, however, high-performance applications such as web servers [79] tend to use distributed architectures in order to cope with scalability and heterogeneity demands. Other distributed monitoring environment exist, for example DIMAPI, developed in the Lobster project [80]. However, here the focus is not using the distributed system to handle extremely high link-rates. Rather, DIMAPI is geared towards obtaining measurements from multiple nodes on a larger network.

The distributed architecture that was introduced in Chapter 5, which aims to process traffic at high link rates, is again shown schematically in Figure 6.1. It consists of heterogeneous processing nodes (N_0, N_1, ...) interconnected in a hierarchy. The first nodes of the processing hierarchy perform easy tasks such as splitting the main 'network traffic' into substreams, while the leaf nodes run the effective traffic processing tasks for the particular substreams (e.g., searching for virus signatures in UDP or TCP traffic in an IDS). Moreover, our architecture allows traffic processing and splitting tasks running together on the same node; a case often met in nodes at the intermediate hierarchy levels.

Our model for dealing with high link rates is to split incoming traffic successively until the stream is sufficiently thinned to make it processable by the end-nodes. While other topologies are certainly possible, a hierarchical, tree-like configuration is a fairly natural fit for such a scenario.

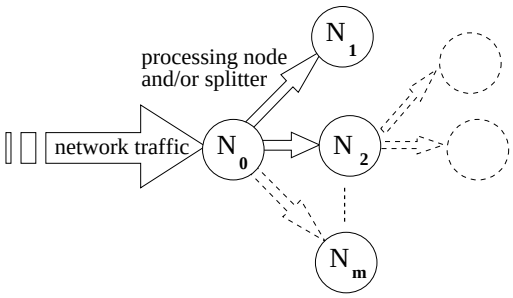


Figure 6.1: Distributed Traffic Processing system.

However, when traffic increases beyond a node's capacity, the system may benefit from offloading the congested node by re-mapping some of the node's tasks onto another node. This re-mapping could consist of either *moving*, or *replicating* the overloaded task onto another node. The former would involve re-routing the traffic from its congested node to the newer node. The latter action would involve splitting of the traffic between those two replica nodes. In addition to the congestion case caused by the traffic we also want to address the external environment changes such as a node failure (hardware failure or software crash), or addition of a new node (e.g., an upgrade). Solving such a failure requires the redistribution of the affected application's task(s) onto the new configuration. Therefore, when the environment changes (e.g., bandwidth increase, system failure, or system upgrade), a control system is needed that takes coordinated adaptive decisions in order to stabilise the perturbed system. We want the control system to be automatic and unmanned because it services non-stop systems that work at high speeds and fast environment changes and hence, fast decisions need to be taken.

This chapter, presents a first step in solving these problems: Control architecture for Distributed Traffic Processing systems (CONDUCTOR). In the CONDUCTOR architecture (see Figure 6.2), the controller monitors the process by means of system workload, and compares the measured states to a process model. Next, it computes a control decision to correct or even to prevent undesired system behaviour like hardware failure or congestion, respectively. Building CONDUCTOR we use control theory guidelines as follows: we first study the process behaviour in order to identify the process model in Section 7.1.1, then we design a controller that suits the specific process model in Section 7.1.2, and finally we simulate the entire control system to evaluate system stability and controller efficiency in various environment changes in Section 7.2.

The CONDUCTOR builds on three research domains: control theory, resource management and network traffic processing. We stress that we have only taken a first step towards solving this quite complicated problem. More than anything we are interested in exploring the problem area. As far as we know, we are the first to look at this problem in the field of distributed traffic processing.

A theoretical design using a hierarchy of controllers that controls a multiprogrammed parallel system was introduced by Feitelson [81]. Such a control structure, proposed for the development of operating systems for parallel computers, allows dynamic repartitioning according to changing job requirements. Although their design goals target the development

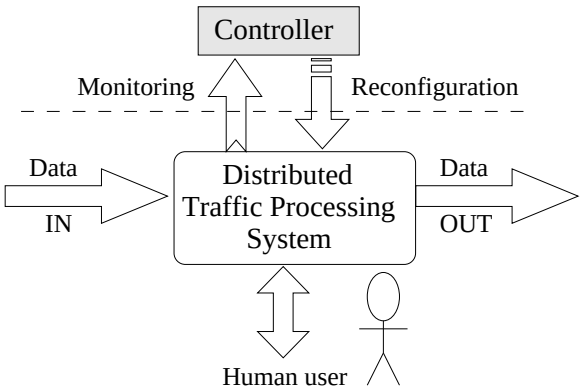


Figure 6.2: CONDUCTOR architecture.

of operating systems that provide dynamic resource partitioning of a parallel computer to user's applications, we noticed a few ideas applicable to our distributed traffic processing systems. The control of resources in a distributed system can be centralised or distributed, or a combination of the two. We have mentioned earlier that organising our distributed processing architecture in a tree-like hierarchy (as illustrated in Figure 6.1) seems a 'natural fit' for our problem domain. One way of dealing with control would therefore be to also distribute the control system in a tree-like hierarchy that mirrors the hierarchy of processing nodes. Alternatively, considering the relatively small number of nodes, a fully centralised solution may be a good fit for most practical use-cases.

The use of control theory in software systems is successfully applied in many computer science fields. The most relevant to our distributed traffic processing system are Varaiya's work in traffic control on automated highway systems [82] and Patterson's work in control of scalable event processing systems [83]. Although a traffic control on automated highway systems presents different research problems than the control of our network traffic processing system such as the problem of optimising of some criteria, we meet several common research questions such as the problem of system stability when the environment changes, or the problem of laziness when there is always a delay between a decision and the effective execution of the decision. The control of an event processing system that uses parallelism to provide scalability, needs to address the following problems, as shown in [83]: (1) provide flow control within replicated elements, and (2) balance the load in the presence of variable processing demands and other disturbances. Besides the event processing, our traffic processing targets simpler goals: to keep the system stable by offloading the congested nodes when the environments changes.

In resource management, current trends look for control of intensive computation systems like web-servers [84], highly-available database farms [85], and grid computing. Despite the limitation of only three tiers (a web-server, an application server container for web-services, and a database) of the proposed control of web-servers, we see an adaptive control algorithm such as proposed in [84] suitable in a centralised control of a distributed traffic processing system. The adaptivity property comes from the ability of the control system to learn from past behaviours. We also look at ways to move jobs for load balancing. Some of this approach

was influenced by work in Grid Computing . For example, the work of Du *et al.* [86] presents a dynamic scheduling algorithm for heterogeneous distributed environment that takes into account a migration cost which is estimated based on statistical data provided by a monitoring system. Tantawi and Towsley [87] tackle the problem of balancing the load on a distributed heterogeneous system using queuing models for hosts and networks.

Most research in network traffic control, such as admission control in the presence of traffic changes, rely on predictors built on traffic models such as Markov, or regression [88], and therefore, these solutions are dependent on the traffic characteristics. These admission control systems are measurement based (MBAC [89–92]), experience based (EBAC [93]), or prediction based [94]. However, since traffic patterns on the network may change rapidly, we do not have time to understand and model it. Numerous studies show that data traffic in high-speed networks exhibits self-similarity [95, 96] that cannot be captured by classical models (e.g., Markov and regression models), hence self-similar models were developed (e.g., FARIMA, GARMA, MMSE) [97]. The problem with self-similar models is that they are computationally complex. Moreover, their fitting procedure is very time consuming while their parameters cannot be estimated using the on-line measurements. Although CONDUCTOR does rely on on-line measurements, it measures the system *workload* rather than traffic and makes control decisions without any traffic analysis. In our architecture, a system workload represents a metric of how much a node is loaded by the processing tasks and is given in %: 0 % when it is not loaded at all, and 100 % means that it is fully loaded and became congested.

6.2 Architecture

In CONDUCTOR, as in most distributed systems, the basic 'processing element' is simply called a *node*. A node is a piece of hardware interconnected to other node(s) and runs multiple tasks (software components written especially for traffic processing). Our heterogeneous distributed architecture uses various hardware implementations for nodes (e.g., commodity PC, and specialised silicon like network processors). It might happen that some nodes have hardware support for parallel tasks. For instance, a network processor can use its cores for running multiple tasks in parallel. In this case we can say that each task runs on its 'core' (hardware core). However, in the case of a node made of a commodity PC, we say that the node has only one core, a general purpose CPU, that supports multiple tasks in a time-sharing architecture.

We illustrate a typical case of task re-mapping in the next section, then we describe the CONDUCTOR architecture components, followed by the controller synthesis, and end with the full architecture description.

6.2.1 Task re-mapping

Suppose we have a traffic processing hierarchy as shown in Figure 6.3.a. For the sake of simplicity, in this section we assume that each node supports only one task and that some nodes offer more processing power than others (e.g., nodes B and F have three times as much processing capacity than the others). We will show how we handle truly heterogeneous configurations later. Nodes A, C, D, E and G are fully loaded, node B is only half-loaded,

and node F is empty (fully available for traffic processing tasks). We also have two existing routes of traffic type 'TCP' that are replicated in order to feed the nodes B and E with packets for processing. Suppose that node E becomes congested. We see that there are two nodes with available resources, as follows: 100 % on node F (empty) and 50 % on node B (half loaded). Although, at first glance, one might think that node F is the best option to re-map the overloaded node's task because it is 100 % available, we need to analyse carefully the entire moving effort including other parameters like re-routing of the traffic that is processed by the overloaded task (see Figure 6.3.b).

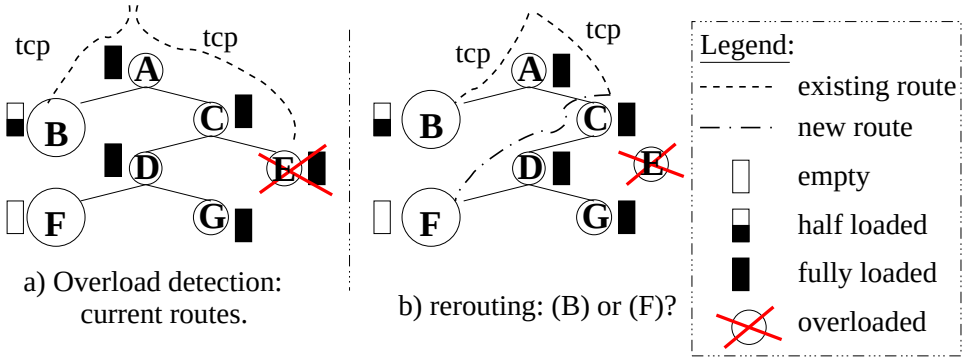


Figure 6.3: Task re-mapping and traffic re-routing.

Notice that we say 'a node is congested', and hence is in trouble, when it has at least one task that gets overloaded. In other words, the 'troubled' task cannot keep up the processing speed with the incoming traffic bandwidth. In our case, a task becomes overloaded when its workload reaches a *programmable* threshold expressed, for instance, by a workload $W >= 90$ %. A programmable threshold allows us to set lower levels than the effective congestion level 100 % in order to give to the control system enough time for offloading the task before it becomes effectively congested. The 'overloading' threshold is determined by the controller according to some specific parameters of the system such as the buffer size of the parent and it is described later in this chapter.

As illustrated in Figure 6.3, when a node gets congested (e.g., node E) one needs to re-map the node's task(s) such that it offloads the overloaded task. Because in our example we assumed that each node supports one task, node E needs to re-map to another healthy available node (e.g., node F or node B). The task re-mapping from a troubled node (source) to a healthy node (target) involves the following aspects:

- moving or replicating: offloading a congested node could be done either by moving the overloaded task to another node, or by replicating the overloaded task onto another node;
- multiple tasks support in a node: when using special hardware that supports multiple tasks (e.g., one task per each hardware-core of a network processor), one can re-map an overloaded task from a congested node onto the same node but on a different core;
- node heterogeneity:

  - is the source node's task available – in terms of hardware and software compatibility – for the target node? For instance, when using different hardware and software platforms for nodes then we need to have the task compilable for every platform;
  - will the target node process the troubled task better or worse than the source node?

- state: has the source node local state (e.g., packet counters, hash tables) that needs to be transferred to the target node? When moving a task together with the state information we need to apply a specific transfer strategy so that it prevents packet loss or corrupted processing results;
- graph dependency: has the source node any children dependent on its traffic? If so, do we move the dependent children together with their parent or do we offload the parent node by moving another independent task from the troubled node (having no children dependency) and then replicating locally the troubled task onto the same node.

Besides the main *traffic processing* role, a node has two other roles: *traffic splitting* and *workload management*, as described in the next sections. Across all these three roles, we build a control path that allows a supervisor to coordinate the entire distributed system.

6.2.2 Traffic splitting

As shown in Chapter 5, the main goal of traffic splitting is to accommodate the incoming traffic in a *node* in order to prevent node congestion and/or keep the node busy in a healthy state. Sometimes we need to split the traffic multiple times (successively at multiple levels) in order to reduce it sufficiently for each processing node's capacity. From a practical point of view, we build a distributed traffic processing system by exploiting hardware usage at various levels in a processing *hierarchy*: high speed packet processing systems (e.g., programmable routers, network processors) near the root and more application-oriented specialised systems near the leaves. Although the traffic splitting architecture theoretically can use any network topology (e.g., star, mesh, B-trees), we have chosen a hierarchical tree-like topology because it is simple and meets all the requirements mentioned earlier: successively splitting needs and hardware usage in a distributed fashion.

The basic splitting principle is shown once more in Figure 6.4. As described in Chapter 5, Node A splits the main network traffic in substreams according to a programmable routing policy implemented within the FPL program with the help of the SPLIT/TILPS extensions for the *NET-FFPF* distributed processing system. In our control architecture, CONDUCTOR controls the way the traffic flows in the distributed traffic processing by re-programming the task in each processing node which also splits traffic. Moreover, CONDUCTOR controls the re-mapping of the tasks on the *NET-FFPF* as we shall see in the next sections.

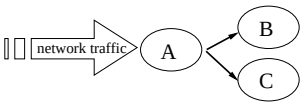


Figure 6.4: Traffic Splitting.

Conductor

CONDUCTOR is a software component designed for control of the distributed traffic processing system. Although in principle this control module can be distributed across the existing distributed processing system, for the sake of simplicity we present in the next sections a centralised control architecture and hence, CONDUCTOR is located in a supervisor host. CONDUCTOR controls the system behaviour by handling the tasks (moving or replicating) across the distributed nodes and routing the traffic accordingly. As mentioned earlier, a *task* is a software component that performs processing of certain *traffic flow* with the support of a hardware *core*. In our architecture, a node's flow of traffic is a subset of the main incoming traffic stream as split and filtered by all the ancestor nodes in a processing hierarchy. In other words, a flow is a stream of packets which may have some common characteristics (e.g., all packets are UDP), or which may have nothing in common (e.g., they were sampled by a parent node 1 in 100).

In order to control the distributed system behaviour, CONDUCTOR provides the following features:

- CONDUCTOR controls the traffic routing based on the programmable routing policy using any suitable packet criterion (e.g., small versus large packets, certain flow identifiers, or sampling). For instance, when CONDUCTOR sets a new traffic route on a certain node (e.g., node A), the split will generally be based on a new SPLIT/TILPS section in the FPL program. When CONDUCTOR needs to re-route an existing flow (e.g, from $A \rightarrow B$ to $A \rightarrow C$) then it only rewrites the MAC addresses on node A such as to point from B to C, as described in Section 5.3.

- CONDUCTOR controls the mapping of tasks onto the distributed nodes by evaluating each node's load from the processing hierarchy and, when needed (e.g., a node gets congested because of an overloaded task), it decides whether to move or replicate a task from a congested node to another node. To do so, CONDUCTOR chooses, from the processing hierarchy, a node that has an available core based on a cost function (C) for moving a task from its node (*source*) to another node (*target*). The cost function returns an evaluation of a task moving effort between two nodes in a traffic processing hierarchy: $C(source \rightarrow target, flow_X)$.

Summarising in Figure 6.5, from the control theory point of view, CONDUCTOR is the *controller* that controls the *process* – entire distributed traffic processing system – in order to keep the system stable regardless of the environment changes.

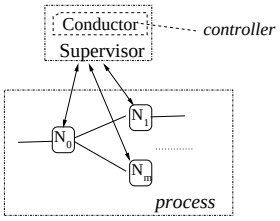


Figure 6.5: A centralised controller controls the process.

6.2.3 Traffic processing

As illustrated in Figure 6.6, a *Node* has multiple processing tasks (`T`) that process traffic in parallel and a *workload manager (*WorkMan*)* that monitors and controls the tasks.

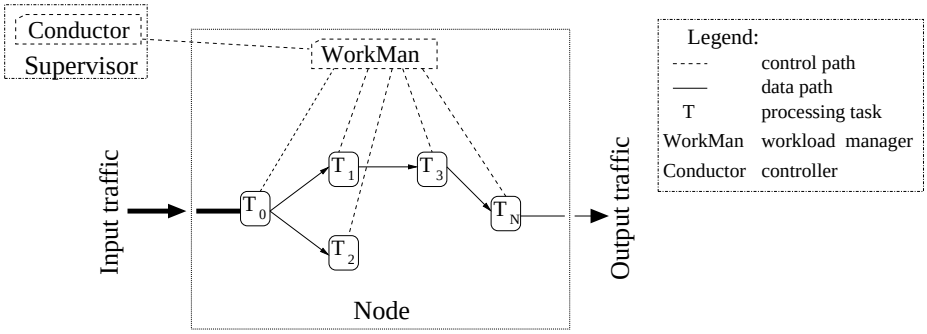


Figure 6.6: Traffic processing in a 'Node'.

For the purpose of control, in a distributed processing system, each node is monitored and controlled by a supervisor. Specifically, CONDUCTOR monitors the load on each node through the interface provided by *WorkMan*. *WorkMan* is a software module that offers two required functionalities for interfacing a node with its supervisor. First, it provides the node's status to the supervisor. Second, it controls the tasks of the node. In our context, a node's workload is computed by the *WorkMan*, adjusted to a suitable metric, and passed to the supervisor as a load status *L*. The details of this computation will be discussed in Section 6.2.4. When needed, the *WorkMan* can point the supervisor to a troubled task that congests the entire node in order to offload it. For instance, a node is congested even though it still has unused cores.

The workload (of a node `Node[i]` or a task `T[j]`) is quantified as follows:

$$L = \begin{cases} -1 & \text{failure} \\ 0 & \text{unused} \\ 1-100 & \text{in use (load value)} \end{cases}$$

The failure status is set by the *WorkMan* when it detects packet loss because of a certain task, or is set by the CONDUCTOR when the connection to a certain node went down. The former case may happen when, despite the control actions to offload the node in time without any packet loss, the *WorkMan* detects packet loss in the troubled task. The latter case may happen when a certain node does not answer to a heart-beat mechanism of the CONDUCTOR because of a hardware or a serious software problem. The heart-beat mechanism involves a periodical short message exchange between CONDUCTOR and each node. When CONDUCTOR does not receive an answer from a certain node to its heart-beat message within a time out, then it tries again for another two times. If there is no answer at all from the troubled node, then CONDUCTOR sets the status of the troubled node to 'failure'.

Although one node may be composed of heterogeneous hardware sub-components such as a control processor, μEngines, and eventually a host PC, for the sake of simplicity, we assume that on a single node all cores are homogeneous. In other words, we address the

heterogeneity at the level of the distributed architecture composed of nodes based on different technologies.

6.2.4 Resource accounting

Supervising nodes requires measuring the workload on a given node's resources. Although our measurement problem refers to a specific system – parallel traffic processing – we formulate this problem using general and known principles based on a single producer/multiple consumers communication buffer.

In Figure 6.7 our producer/consumers problem is drawn. By design, the producer stores the input traffic in a large shared circular packet buffer (*PBuf*) and advances the single write pointer (W) slot by slot as new traffic arrives. At any moment, all tasks may look for new packets by means of their read pointers (R_{Task_i}) until the last written packet pointed to by the common write pointer (W) has been processed. However, the 'producer' never overwrites the reader's buffer space and when *PBuf* is full, all new arrivals are dropped. Therefore, here we can measure only a global (per node) packet loss.

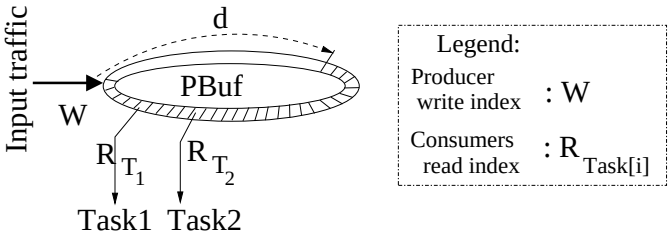


Figure 6.7: Traffic processing model.

Given a *Node* architecture as described in the previous section, a workload evaluation requires detailed measurements of each hardware and software component involved in traffic processing. For instance, in order to evaluate each task's workload we need to measure the processor's load, the memory size in use, the load on the buses, etc.. Moreover, estimating the next system state would require modelling the traffic in the controller. Instead, we reduce the multi-dimensional resource workload measurement problem to a simple *load status* unit as a function of the *buffer usage* percentage expressed by the distance `d` between the producer and the slowest consumer, as illustrated in Figure 6.7:

$$d = \frac{W - R_{slowest}}{PBuf\_Size} * 100 \quad [\%]$$

6.2.5 Resource screening

Re-mapping tasks across the distributed nodes also requires choosing a best fit target node from the processing hierarchy. By design, CONDUCTOR handles the re-mapping of tasks with the help of a cost function. The cost function returns an evaluation of a task moving effort between two nodes in a traffic processing hierarchy: $C(source \rightarrow target, flow_X)$.

In our context, the simplest cost function sounds as follows: "search for a node with an available core with appropriate capacity that will require its routed traffic to pass the smallest number of hops from the root node in the processing hierarchy". However, in a large

distributed system, one might meet multiple choices for the same number of hops such as multiple target nodes with an available core. Then, do we re-map the troubled task on the node with the largest amount of available capacity or one that, capacity-wise, offers the best fit? Indeed, we may follow any of three strategies: first-fit, best-fit, worst-fit. Therefore, in practice, the cost function gets more complex as it needs to take into account multiple parameters such as capacity of the target node's core to run the troubled task, link load to support additional routed traffic, etc.

Notice that the cost function needs to choose an available core that will support the overloaded task. In other words, the cost function must know, at offloading time, how well the overloaded task will run in the available core. The simplest case is to know that the task will perform at least "better" and not worse. Therefore, we need a profiling mechanism that estimates the performance of the task on a new core.

Task profiling

Task profiling provides an estimation of how well a task performs in a certain core. Practically, task profiling consists of starting a task in an available core of a certain type, routing of some traffic for the purpose of feeding the profiled task, running the task for a short period of time, recording the task performance for the tested core, and discarding the task processing results.

Profiling of a task can be done at two moments during the system lifetime: on demand, or in advance. The former case happens when a task becomes overloaded and it needs to be offloaded quickly. The latter case is supposed to have the profiling done before the offloading time. For instance, at system runtime, the controller profiles each running task from the processing hierarchy onto every different core type that is available. The profiling takes place on a definite period of time (e.g., 10 seconds) and the supervisor marks the performance of the profiled task on the specific core on a two dimensional table: task versus core.

We can see that both profiling methods have advantages and disadvantages. On the one hand, running the profiling *on demand* is much simpler than the second method because it runs only when a certain task is overloaded and it profiles only on a few available cores until it finds the core with appropriate capacity. Besides the advantages, the on-demand profiling has a major drawback: it takes much and may lead to packet loss during task re-mapping (as re-mapping may only start when the profiling is done). On the other hand, using *in advance* profiling is more complex than the former because we need a scheduler that tracks all the running tasks and all available cores and performs on-line profiling with as little intervention in the run-time system as possible. The major advantage is that at any moment, during system lifetime, the controller has a more or less view over the available cores and their capacity to run tasks.

A simple example cost function is expressed in Equation 6.1. In our example, the cost function depends on the existing routes for the relevant traffic flow for the re-mapped task onto a target node. The resource availability on the target node is a separate decision and may follow any of the first-fit, best-fit or worst-fit strategies.

$$C(source \rightarrow target, flow_X) = \sum_{n=root}^{target} Link_{n,n+1}(flow_X), \tag{6.1}$$

where:

$C(source \rightarrow target, flow_X)$ is the cost of moving a task from source node to target node and that processes traffic $flow_X$. The function returns an integer value,

$$Link_{n,n+1}(flow_X) = \begin{cases} 0 & \text{if the link already carries the traffic } flow_X \\ 1 & \text{if the link needs to support a new traffic flow} \end{cases},$$

the total cost of this sum increases with one unit for every link that needs to support a new traffic flow along the routing path from root till the target node.

Note however, that this is an example and different cost functions can be plugged in, or existing ones can be refined according to new topology or future technology. A more advanced cost function could also quantify, for example, system parameters like local state to transfer together with the task, whether the troubled task has children in the processing hierarchy or not, the available bandwidth on each link that needs to support the new traffic routed for the moved task, etc.

6.2.6 Resource control topologies

Using a bottom-up design, we develop a high-level abstraction of a low-level system. Control is applied on the top level with the idea that the low level will perform accordingly. Control of a hierarchical system can be *centralised* or *distributed*. One may identify three control algorithms ranging from a centralised to a fully distributed system as shortly described below.

- In *centralised control*, a *supervisor* controls the whole system by administrating the entire physical distributed topology as well as the load status of each node.
- In *federated control*, a node has supervisor rights over the nodes beneath it in the processing hierarchy.
- In *distributed control*, there is no supervisor and no control relationships between nodes. For instance, a failing node may broadcast an 'S.O.S.' message to the entire distributed control network and then listen for answers. Then each node evaluates the 'S.O.S.' message request and when it finds enough available resources will reply to the 'S.O.S.' owner. In the end, the failed node re-maps its failing task to the best fit node (in case of multiple 'S.O.S.' answers).

In this thesis we will only investigate centralised control (in the next chapter) because it is the simplest one to be implemented. One could say that if centralised control cannot be successfully implemented then distributed control will be even harder to be implemented.

6.3 Summary

In this chapter, we saw that distributed systems require, at a minimum, the following three functions: (1) mapping of the user applications onto a distributed environment (presented in Chapter 5), (2) distributed processing in the runtime (also presented in Chapter 5), and (3) result retrieval from the distributed environment (already addressed in the field of distributed

systems, e.g., by Salton and Callan books [98, 99]). Moreover, when a distributed application works in a dynamic environment and assuming that the resources are limited, the system itself needs to (4) adapt to the environment changes in order to remain stable.

Chapter 7

A Simple Implementation of Conductor Based on Centralised Control

The previous chapter presented the CONDUCTOR architecture. As a short summary, the CONDUCTOR architecture provides the following features: traffic processing, traffic splitting, and workload management. As a result, we build CONDUCTOR on the following components that all interact: the *processing node* (*Node*) for traffic processing, the *workload manager* (*WorkMan*) for resource accounting in every node, and the *Conductor* for system control by means of traffic routing and task re-mapping. Using these components, for the proof of concept of adaptive control, we built only the 'centralised' topology, which we will describe in this chapter. We also stress that this is a first attempt in which we deliberately simplified the very complex problem in order to be able to evaluate the main principle. It should not be seen as a full or mature solution.

Figure 7.1 depicts the 'centralised' case of CONDUCTOR. Given a distributed traffic processing system composed of a node hierarchy ($Node_{1.1}$, $Node_{2.1}$, etc.), we add a supervisor that monitors and controls the behaviour of the distributed traffic processing system with the help of workload managers (*WorkMans*) plugged in each processing node ($WorkMan_{1.1}$, $WorkMan_{2.1}$, etc.). The supervisor targets two global system goals (stability and performance) by the following types of control operations: (1) (re-)mapping of processing tasks onto the distributed traffic processing hierarchy and (2) (re-)routing of traffic streams. The first control type involves moving or replicating processing tasks from a congested node to a more lightly loaded node in order to prevent an effective node failure. The second control type comes as a need of the first: moving a task in different position in the processing hierarchy often requires re-routing of the traffic feeding the task. Both control types work together in order to assure *performance* and *stability*.

Figure 7.1 shows an example of centralised control of a distributed processing architecture in a hierarchical topology.

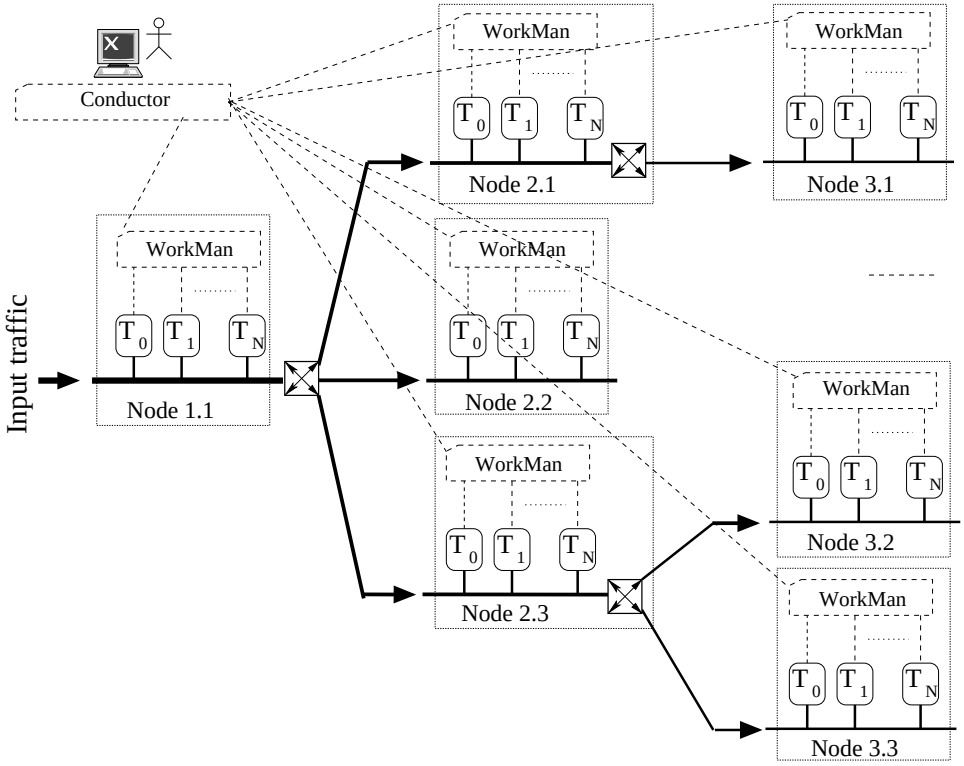


Figure 7.1: Centralised control of distributed processing architecture.

The distributed traffic processing system works as follows: the main stream ('input traffic') is first processed in $Node_{1.1}$ and primarily involves traffic splitting rather than traffic processing. Then, the split traffic (sub-streams of the main stream) flows in the next nodes of the processing hierarchy ($Node_{2.1}$, $Node_{2.2}$, $Node_{2.3}$). Some of these nodes are end-nodes (e.g., $Node_{2.2}$) that perform traffic processing only and others are also splitters (e.g., $Node_{2.1}$, $Node_{2.3}$). When going down in the processing hierarchy we increasingly meet traffic processing nodes and fewer splitters. We assume that the end-nodes are properly chosen (in terms of hardware capacity for the incoming traffic stream) by the initial mapping according to the FPL program. For instance, $Node_{2.2}$ is able to process the traffic split by the $Node_{1.1}$ (one step splitting), while $Node_{3.2}$ and $Node_{3.3}$ need two splitting steps.

In our example of distributed architecture illustrated in Figure 7.1 we can also see that we used two types of switches for traffic splitting: stand-alone switches such as the one located after $Node_{1.1}$, or built-in switches such as the ones in $Node_{2.1}$ and $Node_{3.1}$. Technically speaking, the former can be a standard Ethernet switch and the latter is a processing node with multiple output ports available. Supporting both switch types, stand-alone and built-in, provides flexibility in building a distributed traffic processing system by interconnecting of the best price/performance nodes.

So far, we have seen how the distributed traffic processing system works and how the

control is layered on top. A high-level view of CONDUCTOR is as follows. Suppose that $WorkMan_{2.2}$ (see Figure 7.1) detects that its node ($Node_{2.2}$) starts being congested because of the increasing of the incoming stream. Then, the supervisor identifies which task is overloaded on $Node_{2.2}$ and start searching for an available core that could offload the troubled task (e.g., $Task_4$). Suppose the supervisor finds an available core on $Node_{3.3}$ that, according to the profiling data, is suitable for handling the troubled task ($Node_{2.2.Task_4}$). Then, the supervisor takes care of loading the troubled task's code onto the chosen core of $Node_{3.3}$, of starting the target task and finally of re-routing the corresponding traffic stream from $Node_{2.2}$ to $Node_{3.3}$. An important challenge is to make the transfer as non-disruptive as possible. Section 7.1 explains in detail how we implemented the transfer so as to minimise the number of packets being dropped.

7.1 Centralised control for adaptive processing

In general, the problem of control can be described as follows: given a dynamic system, or a collection of interacting dynamic systems, synthesise a *controller* so that the system satisfies some design specifications. In other words, the controller reads/measures the process states (system workload), it compares them to the process model, and computes a control decision to correct or prevent undesired system behaviour such as congestion.

The control system is responsible for detecting any overloads, underloads, or hardware changes (e.g., a dead node) and it tries to redistribute the workload by re-mapping of the affected tasks over the rest of the distributed system.

In order to synthesise a controller we need to identify the process behaviour, find the model spaces, design the control loops, choose the best-fit controller types for the specific design space and finally tune the controller(s) by simulation or real experiments.

7.1.1 Model identification in a distributed traffic processing system

Given a processing node as described in Section 6.2.3, we first need to identify the process model by investigating its behaviour, then try to synthesise a controller that will control the real process.

Processing node behaviour

A node's workload is expressed as a buffer usage level percentage. A workload example in a processing node is depicted in Figure 7.2. There is a maximum level (MAX) up to where the system is able to process packets. In other words, when the system hits the MAX level it means it gets congested and starts dropping packets. The HI level is an intermediate threshold level that is set-up by the supervisor as a reference so as to provide enough provisioning time for solving the problem of re-mapping of the task. Therefore, HI is estimated at deployment time and depends on the size of the buffers and the estimated core capacity to run the task in normal environment conditions. As we will see, it will be dynamically adjusted at runtime. Moreover, when the system workload is below a minimum level (MIN) it means that the node is underloaded and its task(s) should be moved to other nodes so as to free up the underloaded node completely. The underloaded node informs its supervisor about its new

state and the supervisor decides, according to a 'defragmentation policy', how and when to re-map the tasks from the underloaded node so as to efficiently use the resources of the entire distributed traffic processing system.

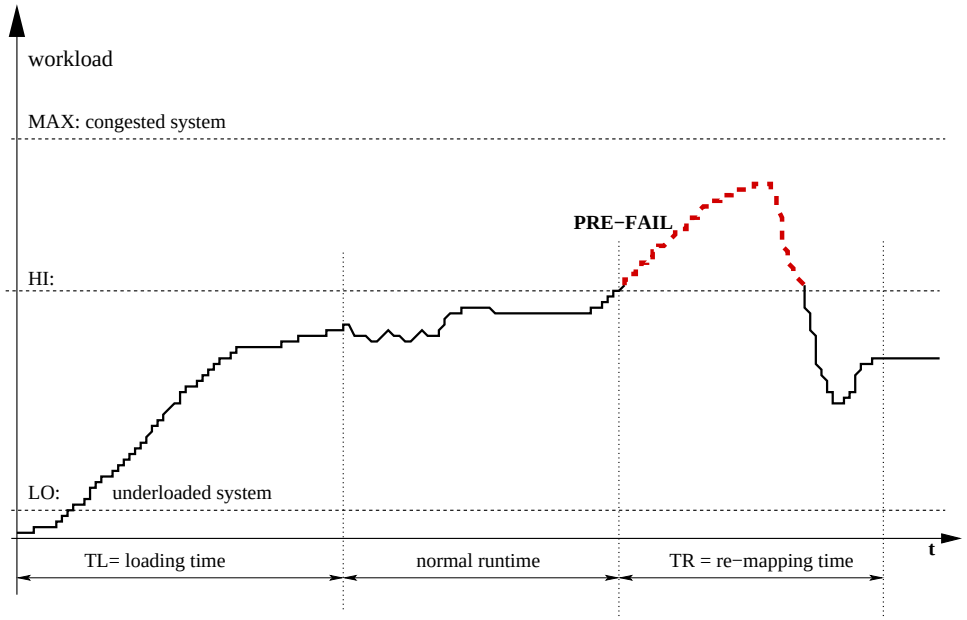


Figure 7.2: Dynamic behaviour in a processing node.

Let us considering the example in Figure 7.2. When the system starts up, it takes a while, called loading time (TL in Figure 7.2). During the loading time the system is filling all the hardware cores with tasks and the supervisor is achieving the first stable state. Then we say that the entire system is stable as long as each node's workload is steady or slightly fluctuating up to node's HI level. However, the workload could go up to the MAX limit because of changes in the environment. When the workload exceeds the HI reference, the supervisor throws an exception `PRE-FAIL` in order to take an early decision for solving the problem before the node reaches the MAX limit. Next, the system tries to solve the problem by re-mapping the troubled task from congested node to an available core with an appropriate capacity. After the re-mapping time, TR, the system comes back to a stable state.

Model spaces

In order to identify the process parameters (TL, TR, etc.), we need to do some practical experiments. We take two tasks located on different nodes: $Node_1.Task_x$ and $Node_2.Task_y$ and work on the following scenario (see Figure 7.3): $Task_x$ gets overloaded in $Node_1$, and we note traffic re-routing from $Task_x$ to $Task_y$, and task moving from $Node_1$ to $Node_2$. Although the experiments use, for the sake of simplicity, tasks with no dependent children, the measured parameters can be scaled according to the amount of children that would need to be re-mapped together with their parent.

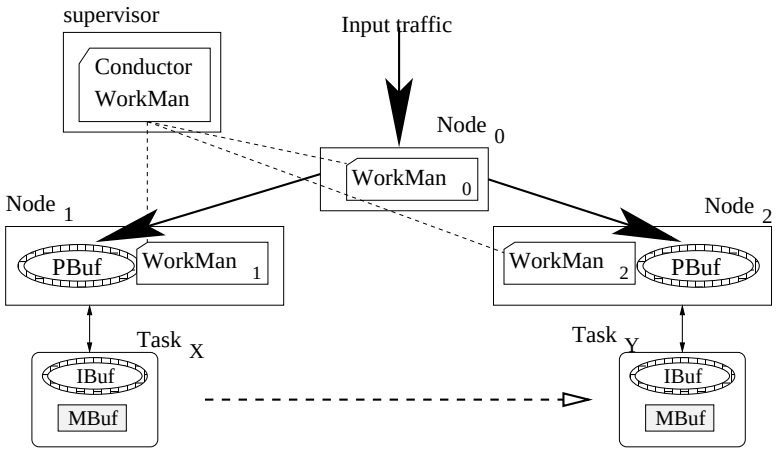


Figure 7.3: Moving $Task_x \rightarrow Task_y$.

The algorithm of a task moving between two nodes tries to solve the following two problems: (1) transferring of process state and (2) minimise the packet loss. Let us assume that when the supervisor receives a 'congestion' notification from $Node_1.Task_x$, it first prepares the offloading of the overloaded task. It finds an available core that has the appropriate capacity to run the troubled task, then it copies and loads the object code onto the core without starting the task effectively. Next, it re-routes the traffic from $Node_1$ to $Node_2$. Then the supervisor waits for the 'ready' status from $Node_1$ that tells when the shared buffer *PBuf* is completely drained. In other words, we keep the $Node_1.Task_x$ processing of the remaining packets in its *PBuf* while the new incoming traffic is routed to the $Node_2$ and is currently stored in $Node_2.PBuf$. When the supervisor receives the 'ready' status (all packets from $Node_1$ were processed), it issues a state transfer from $Node_1.Task_x \rightarrow Node_2.Task_y$ by means of an *MBuf* copy. After the state transfer, the supervisor is ready to start the new processing task: $Node_2.Task_y$.

We implemented this algorithm on a distributed architecture (see Figure 7.3) composed of a commodity PC (used as supervisor), a switch, and two different generations of network processors: IXP2400 and IXP2850. The task used in this experiment consists of a simple application that counts the UDP packets in a stateful counter and computes a hash over the first 64 Bytes of the packet. The counter gives us a local state that needs to be transferred together with the task when it gets overloaded. While this application is not in itself particularly useful, it is illustrative for real applications, while also providing a simple constant benchmark. The re-mapping algorithm, in this particular distributed architecture, illustrates a 'system upgrade' case and is described in detail below.

1. The CONDUCTOR calls on $WorkMan_1$ and $WorkMan_2$ to load the same application's code object – 'UDP packet counter and hash over the first 64 Bytes' – on their nodes: $Task_x$ on $Node_1$, $Task_y$ on $Node_2$; then CONDUCTOR starts the $Task_x$ and keeps the $Task_y$ in standby (not running);

2. $Node_1$ classifies all UDP packets to be of interest to the $Task_x$: *IBuf* points to all

UDP copy packets in *PBuf*. We have evaluated various sizes for *PBuf*, as illustrated in Figure 7.4 and described later in this section;

3. $Task_x$ processes all packets from *IBuf* and stores the temporary state/results into *MBuf*. We used a 1150 Bytes long array for storing a hash table in *MBuf*. We found that the size of *MBuf* does not affect the re-mapping time significantly compared to a stateless case when the task has no *MBuf* and hence no need to wait for the drain;

4. A `PRE-FAIL` event occurs in $Node_1$: the incoming UDP traffic grows, exceeds the $Task_x$ reference (HI), and the event is detected by $WorkMan_1$ by means of checking the buffer usage d_x. HI was set at 75 % to ensure with high probability that there is enough time for task re-mapping before the workload would hit the MAX level;

5. $WorkMan_1$ signals the `PRE-FAIL` event to the CONDUCTOR: $WorkMan_1.Task_x = -1$ and therefore, the CONDUCTOR signals $WorkMan_0$ to re-route the UDP traffic from $Node_1$ to $Node_2$;

6. The CONDUCTOR waits for a second event of the troubled node ($Node_1$) that tells when $WorkMan_1.Task_x$ finishes the processing of buffered packets ($Task_x.IBuf$ becomes empty);

7. When the CONDUCTOR receives the second event, it stops the $WorkMan_1.Task_x$ and issues a state transfer: $WorkMan_1.Task_x.MBuf \rightarrow WorkMan_2.Task_y.MBuf$;

8. Finally, the CONDUCTOR starts $WorkMan_2.Task_y$.

The goals of this test are threefold. First, we measure the loading time of an application's code object (a task) on a node, and then on two nodes ($Task_x$ and $Task_y$). This loading time (TL in Figure 7.2) is described below in Step 1. Second, we measure the dead-time between `PRE-FAIL` event occurrence (Step 4) and the successful failure addressing (Step 8) for a single task (no dependent children). This time is depicted in Figure 7.2: TR. Lastly, we find the best way to address the failure detection: a realistic HI threshold in order to signal a `PRE-FAIL` in time for minimising the dead-time introduced by a task movement and avoiding of packet loss.

Our experiments show that the loading time (TL) measured in Step 1 is roughly 1 second. This is mainly due to the fact that we use secure file transfer (scp) for loading the application code remotely. The application's code object file is about 500KB. Moreover, we found that TL is not much dependent on the number of nodes in the distributed architecture due to the simultaneous code loading on all the nodes. The most important limitation is the supervisor's bandwidth for the control link.

When handling a `PRE-FAIL` event, we found that there is a constant detection time of roughly 1.7ms. It involves one way communication over a TCP connection. Next, when moving a task between two nodes we found that the re-distributing time (TR) is dependent on the following attributes:

- feedback communication: we use a dedicated asynchronous TCP connection separate from the main control connection that is needed to inform the CONDUCTOR on each node's status (workload);

- state transfer: the *MBuf* stream size in our case is 1150Bytes and it takes 9.7ms to transfer this state from one node to another;
- draining time: the time needed to process the rest of the packets in *PBuf* since the route was changed and which is therefore, dependent on the *PBuf* size.

Figure 7.4 illustrates the draining and re-distributing (TR) time for different sizes of the shared packed buffer (*PBuf*). The 'simulated drain time' chart was computed using Intel's cycle simulator for the μEngine processing time of each incoming packet. The 'DrainTime' chart was measured locally, in hardware, inside the troubled node using an interrupt-based mechanism for reading the sensors (SRAM memory locations written by the processing tasks). The 'TotalTime' is the entire re-distributing time since the occurrence of the `PRE-FAIL` event until the successful moving of task (Steps 4-8). This time includes the draining time, the copy of local states (*MBuf* stream) and the additional transfer events over the control link (TCP connections).

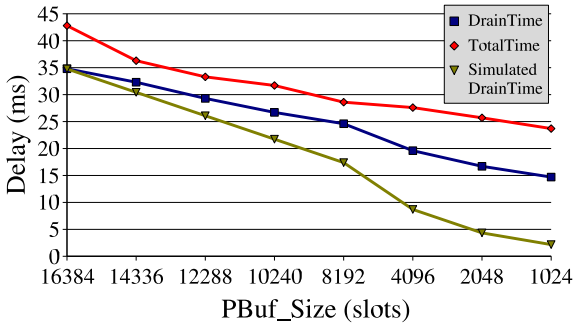


Figure 7.4: Tasks moving benchmark.

7.1.2 Control design

In the previous sections we identified the process behavior and the process model space and now we present the control loop design.

The control loop

For the sake of simplicity, we present in Figure 7.5 the block diagram of a control loop example using only two nodes. Figure 7.5 shows the controller (CONDUCTOR) located on a central supervisor, the actuators ($WorkMan_x$, $WorkMan_y$), and the controlled tasks ($Node_x.Task[]$, $Node_y.Task[]$) located on $Node_x$ and $Node_y$, respectively.

In order to have the supervised system stable regardless of the environment changes, CONDUCTOR has, by design, two sub-objectives: a *continuous analysis* and a *failure analysis*. As the name suggests, the former is performed continuously (at the highest sampling capability of the system), while the latter analysis runs only when a failure event was detected by the first analysis.

In the continuous analysis, CONDUCTOR performs periodically, at the highest sampling rate offered by the system, in our case every 100 msec, two actions. First, CONDUCTOR uses the load status of each node ($L_{Node[x]}$) to update a local *process map*. The process map represents CONDUCTOR's view of the entire system under its control. Note that if the number of nodes increases to a point where scalability becomes an issue, we may be forced to look at a more distributed controller (e.g., a federated one). However, as we assume the number of nodes not to exceed a few tens, we did not explore the distributed control in this thesis. Second, CONDUCTOR checks the evolution of each node's status against a pre-loaded model and sends requests to the failure analysis when, for instance, an overload or underload (or even a hardware failure) problem occurs. A PRE-FAIL event is detected through the evaluation of the model. The model represents a dynamic behaviour of a node as illustrated in Figure 7.2. The model is loaded and set up at deployment time with the system configuration and specific parametric data such as LO and HI threshold levels.

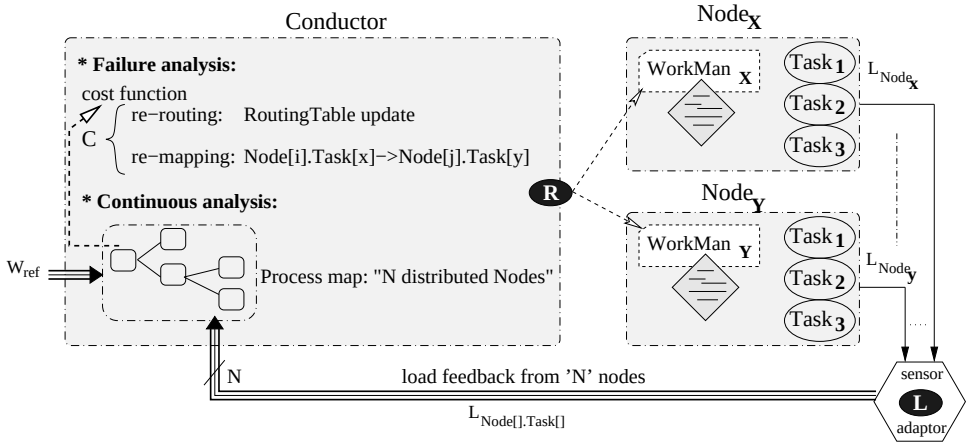


Figure 7.5: Block diagram of the control loop.

The failure analysis processes each PRE-FAIL event received from the continuous analysis. The failure analysis searches for an available core on the same node or another node from the processing hierarchy. When it finds multiple cores available, it chooses the first-fit based on the cost function as described in Equation 6.1. Once such a solution is found (e.g., with the smallest cost function), CONDUCTOR *re-maps* the overloaded task from the congested node on the chosen node with the help of the nodes' actuators (*WorkMans*). This action involves also updating the routing policies (event ® in Figure 7.5) because moving or replicating a task between two nodes requires moving or replicating the needed input stream, respectively.

Another anomaly that the 'failure analysis' can detect is the case when a processing task becomes underloaded; for example, the load status indicates a very low workload value like less than 5 %. In this case, the CONDUCTOR searches through the process map of the distributed system for other available processing nodes that could handle the workload of the underloaded task. When it finds a proper solution, it moves the underloaded task and redistributes the traffic so that the underloaded core becomes available. Then it simply marks

the available core as 'unused'.

Note that the continuous analysis determines an important characteristic of the control systems: whether there could be packet loss during the decision time in case of a failure or whether it can prevent any packet loss. Based on this characteristic, several algorithms are possible for continuous analysis: (1) AIMD, and (2) prediction. We opted for AIMD which stands for additive increase, multiplicative decrease and is a dynamic flow control mechanism used for instance in TCP flow control. Although AIMD will be explained presently, for completeness, we also present a possible alternative below.

AIMD control algorithm

This algorithm is a simple control algorithm inspired by the TCP flow control model. It tries to avoid packet loss during node offloading by adjusting the current HI level according to a previous offload experience so as to provide enough spare time to the system for the next offload experience. Initially, a default value is computed for each threshold: LO and HI. The algorithm starts up using the default thresholds values and then it dynamically adjusts the HI level so as to provide a good tradeoff between early offloading (preventing packet loss) and late offloading (for higher resource utilisation). The algorithm checks, for each offloading experience, whether packet loss occurred or not (see Figure 7.6). The HI level is adjusted according to this answer as follows: when there was no packet loss, the HI level grows one point ($HI = HI + 1$). When there was some packet loss, the HI level decreases according to the formula: $HI = \alpha \cdot HI$, where $\alpha \in [0..1]$. The choice of α is also important because it provides the system's dynamics: how often the troubled tasks are re-mapped. In our experiments we found that an $\alpha = 0.8$ is a good tradeoff value.

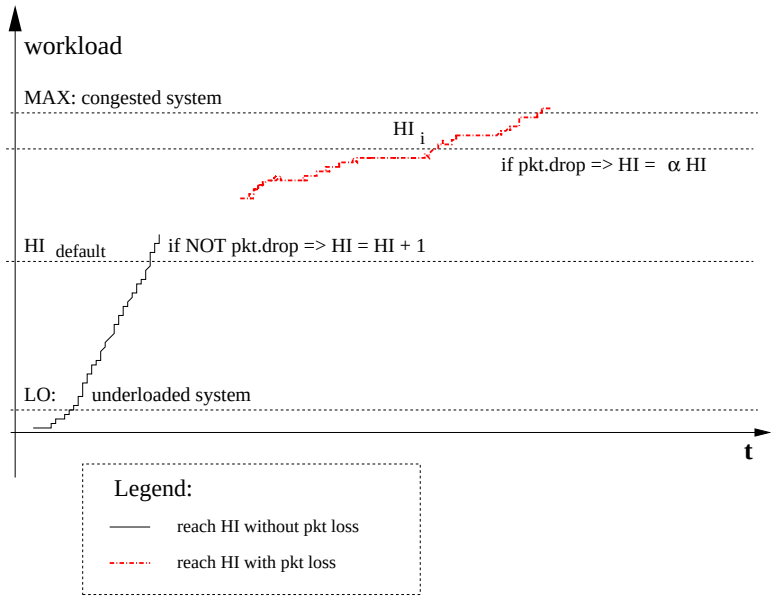


Figure 7.6: AIMD control behaviour.

The advantage of this algorithm consists in its simplicity and the disadvantage is the assumption that packet loss would be acceptable when re-mapping tasks.

Prediction control algorithm

An alternative would be to try and prevent packet loss during task re-mapping by forecasting the congestion moment. A popular approach, largely used in trend estimation, uses *least squares fit* math principles. The efficiency of linear prediction using least squares fit was shown by [100] for traffic congestion forecast of web-servers. Therefore, an implementation of this algorithm is beyond the scope of this research and interested readers are referred to [100].

Summarising, the main job of the controller (CONDUCTOR) is to re-distribute the tasks over the distributed processing hierarchy such as the entire system is well loaded (not underloaded and nor overloaded). A secondary job consists in re-routing of the traffic when re-mapping of the tasks. Although the controller performs a continuous analysis over the load status of the entire distributed system, the jobs run only when the pre-failure event occurs, as shown in Figure 7.2.

7.2 Experiments

In order to analyse the efficiency of the AIMD control algorithm, we first describe a test-bench composed of heterogeneous processing nodes: commodity PCs and network processors. Next, we will present the control behaviour in a scenario of an overloaded task. We will take an example of a simple application as a processing task: pattern matching. In the end, we evaluate and discuss the controller's behaviour when the incoming traffic increases beyond the nodes' processing ability.

7.2.1 Test-bench

Figure 7.7 shows a distributed traffic processing system composed of heterogeneous nodes: commodity PCs (x86) and two network processor generations: IXP2400 and IXP2850. The distributed system uses $Node_0$ to split the incoming traffic between $Node_1$ and $Node_2$. We assume that $Node_0$ does only traffic splitting, $Node_1$ and $Node_3$ run only packet processing tasks, and $Node_2$ performs both functions: packet processing and splitting. On top of this heterogeneous distributed traffic processing system we have a supervisor that supports a CONDUCTOR controller connected to each of the node through a control network illustrated in Figure 7.7 by dashed lines.

Control scenario

In order to show the controller behaviour in case of environmental changes we make the following assumptions. We assume that a task running on a node gets overloaded because of an increasing in the incoming traffic. Consequently, the controller decides to offload the troubled task onto another node. In our example illustrated in Figure 7.7, we suppose to have an overloaded task running on $Node_1$, a commodity PC made of x86 architecture, that needs to

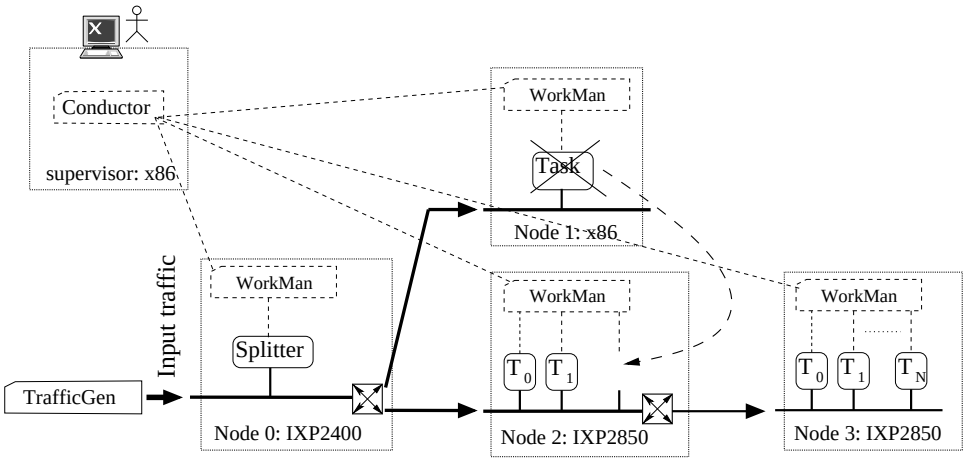


Figure 7.7: An heterogeneous distributed system.

be offloaded onto another node. For the purpose of heterogeneity, we let the controller choose to offload the overloaded task onto a different architecture node: in this case, it happens to choose $Node_2$, an IXP2850 network processor.

Note that in a homogeneous case such as offloading a core to another similar core, we simply need to duplicate the overloaded task onto another available core and then re-distribute the traffic to the duplicated cores.

The main difference in homogeneous environments, it would seem, is that finding an appropriate target on which to offload may be difficult. Unless the core is shared by multiple tasks, or multiple cores share common resources (such as a queue), offloading is pointless and would simply lead to the new core being offloaded. In other words, homogeneous environments limit the options for offloading. On the other hand, homogeneous environments are simpler, as there is no need to establish a relative performance for specific task for each specific core.

7.2.2 Application task: pattern matching

So as to have a task easily overloaded by increasing the input traffic throughput we need a very resource consuming task. We have chosen a task that is both resource consuming and simple: a pattern matching for the entire packet payload. For example, we search for the string ‘witty’ in all the udp packets with the destination port higher than 4000, as the witty virus propagates. We also know that the witty virus places its signature ‘insert.witty’ starting by 146 bytes offset in the udp payload. Although there are better performing searching algorithms such as Aho-Corasick, for the sake of simplicity, we use a linear searching as illustrated in the FPL example given in Listing 7.1.

```
MEM[1];
REG[2];
IF (PKT.IP_PROTO == PROTO_UDP && PKT.UDP_DEST >= 4000) THEN
  R[0] = PKT.IP_TOTAL_LEN-5;                    // save pkt size in register
  FOR (R[1] = 188; R[1] < R[0]; R[1]++)  // start searching from byte 165
    IF ((PKT.B[ R[1]+0 ] == 77) &&
        (PKT.B[ R[1]+1 ] == 69) &&
        (PKT.B[ R[1]+2 ] == 74) &&
        (PKT.B[ R[1]+3 ] == 74) &&
        (PKT.B[ R[1]+4 ] == 79))
    THEN                              // found the signature 'witty'
      M[0]++;                         // increment counter in persistent memory
    FI;
  ROF;
FI;
```

Listing 7.1: Example of intensive FPL application: searching for witty signature.

7.2.3 Controller behaviour

To evaluate the behaviour of the controller we monitor the application task described above while we ramp up the input traffic throughput. First, we start the system with the monitored task mapped on $Node_1$, of the network configuration illustrated in Figure 7.7. Next, the supervisor also detects available cores on $Node_2$ and therefore, it profiles the task on this node. At the same time, TrafficGen generates increasing traffic throughput from 0 to 1000 Mbps during 60 seconds, our chosen duration for this experiment. The generated traffic is routed, according to the control scenario described above, to $Node_1$ which is going to process this traffic. Finally, the entire control scenario is described and illustrated.

Profiling of the task on one μEngine of IXP2850 node shows that the task may cope with the workload given of an input of 1Gbps traffic. We note that this higher performance of IXP2850 compared to the commodity PC is due to features of the parallel processing architecture such as parallelising of the task on four hardware supported threads and reading of memory in chunks of eight bytes. See also Section 2.2 for hardware details.

For the purpose of introducing of environment changes in the controlled system, we need to vary the incoming traffic throughput. In this respect, we ramp up the input rate. Figure 7.8 illustrates the trend of both the generated traffic throughput and the buffer usage measured in $Node_1$ at every second. The generated traffic throughput is not growing linearly due to a limitation of our traffic generator using an older generation of network processor: IXP1200. However, we measured the buffer usage level during a period of 30 seconds in which we see that the evolution is almost constant for low throughput, up to 200 Mbps at second 19th; then the buffer usage grows up very fast. Therefore, we assume that the buffer usage trend on $Node_1$ grows up in an exponential shape and we will illustrate it later, in Figure 7.9, together with the plot of the buffer usage on $Node_2$ after overload.

Notice that the experiment used 'homogeneous' synthetic traffic by means of a UDP stream of constant packet size. The packet size was 250Bytes. However, when using real traffic (variable packet sizes) we expect that the buffer usage would present higher dynamics (oscillations). We could not 'replay' a real captured traffic due to lack of a professional traffic generator that could generate traffic at a variable throughput that is sufficiently high.

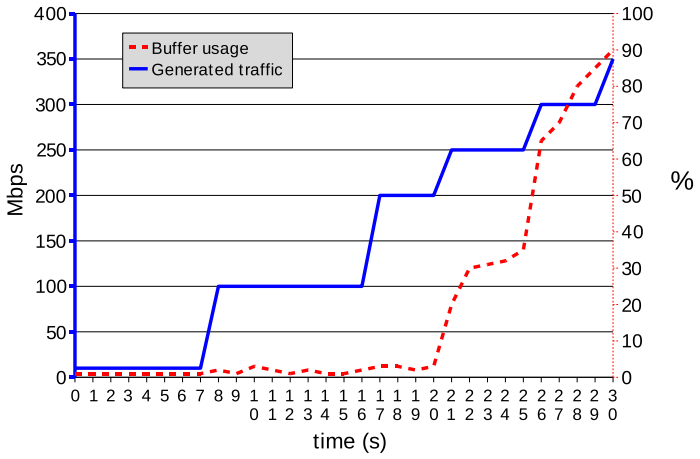


Figure 7.8: The workload measured on $Node_1$.

Figure 7.9 shows the runtime period of the monitored task on $Node_1$ and $Node_2$ on the assumption that the distributed system illustrated in Figure 7.7 receives increasing input traffic. We run the experiment for 60 seconds, because this period is long enough to observe the re-mapping of the monitored task from a congested node ($Node_1$) to another available node ($Node_2$).

Figure 7.9 shows that at time 20 secs, the task's workload running on $Node_1$ reaches the HI level at 70 %. In other words, the buffer usage on this node reaches the workload reference. As a consequence, the control system start offloading $Node_1$ by re-mapping the troubled task to $Node_2$. Due to a re-mapping decision time, the buffer usage level of $Node_1$ still grows up to 85 % until the moment of re-routing of the incoming traffic to $Node_2$. At this moment, the buffer usage of $Node_1$ starts decreasing on the one hand, and we see an increase in the buffer usage of $Node_2$ up to 50 % on the other hand. Notice that the buffer usage on $Node_2$ grows that much (50%) due to a dead-time during which $Node_2$ buffers the packets without processing them until the re-mapping ends. The re-mapping ends when both all the buffered packets from $Node_1$ were processed and $Node_2$ starts the new task. The first is illustrated by the moment when the buffer usage of $Node_1$ became 0, closing the 'draining' period. The second happens when the processing results, a 1150 Bytes array, were successfully transferred from $Node_1$ to $Node_2$. The draining and the results transfer together take about 30 ms and represents the dead-time. The total measured re-mapping time was less than 100 ms and is the sum of the controller's decision time and dead-time. We also notice in Figure 7.9, on the right axis, that the monitored task runs well on $Node_1$ for a traffic input of up to about 280 Mbps.

According to our AIMD implementation, the HI level on $Node_1$ increases with one percent after a re-mapping with no packet loss and therefore, a future task which runs $Node_1$ will be offloaded when its workload reaches the newer HI level of 71%. Besides the simplicity of the AIMD algorithm used in our controller, we note some drawbacks, as follows. The system may loose packets during a re-mapping operation because it does not guarantee

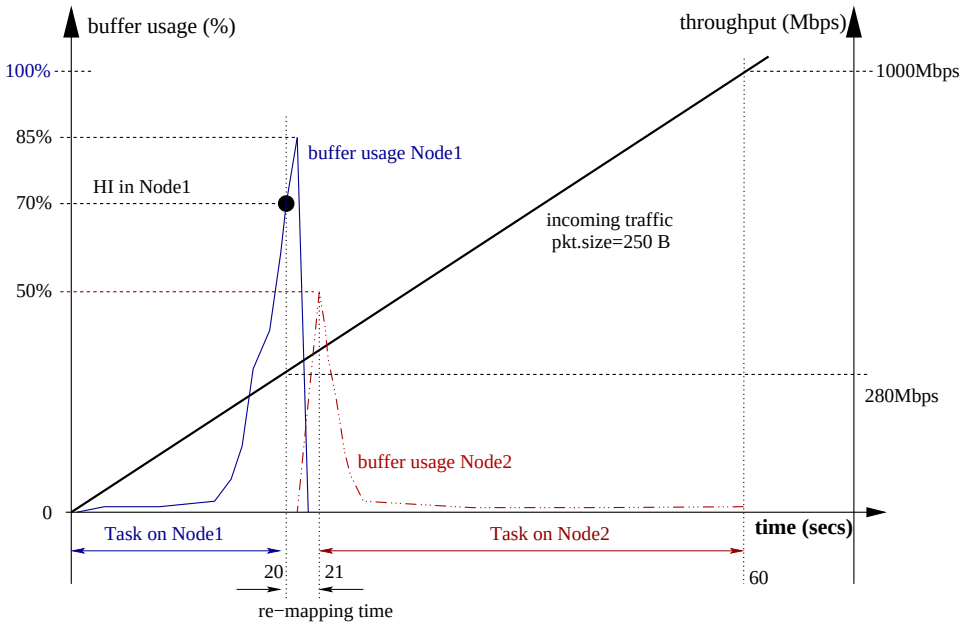


Figure 7.9: The control behaviour.

that the system can buffer the entire input traffic during the re-mapping time. When we set a lower threshold to give enough time for re-mapping then we may have multiple re-mappings that lead to oscillation and hence an instable system. A smarter system may deal with spikes that could temporary overload the node but not congest it.

7.3 Summary

In this chapter, we presented a simple control algorithm implemented in a centralised control topology of the CONDUCTOR architecture described in the previous chapter. We first identified the model of the distributed traffic processing system by evaluating an experiment consisting of a task moving between two nodes. Next, we designed a control loop that will keep the entire distributed processing system stable regardless of external changes in the traffic that may overload tasks. The control loop acts over the distributed system by re-mapping of the overloaded tasks and re-routing of their processed traffic accordingly. Finally, we implemented a simple control algorithm, additive increase / multiplicative decrease (AIMD), and ran an experiment which showed the benefit of using heterogeneous distributed systems for traffic processing. We emphasise again that both the conductor architecture and its evaluation are to be considered preliminary steps towards solving a very complex control problem. For real-world systems, the cost functions, AIMD implementation and various other aspect of the system should be improved. Similarly, experiments with moving more tasks at the same time, and tasks with children are needed. At the same time, we believe that our results show that offloading while minimising packet loss as performed in CONDUCTOR shows promise.

Chapter 8

Beyond Monitoring: the Token Based Switch

So far we have focused on a single application domain, that of monitoring. As a result, much of our attention was devoted to efficient packet reception, processing and gathering of intermediate and final results. Besides monitoring there are several other application domains that need high-speed packet processing. Routing may be the most conspicuous example. However, in routing we do not have many contributions to make as it is well catered to by commercial vendors. Instead, we turn to a new type of application domain that aims to provide certain packets preferential treatment by sending them over fast fibre-optic connections rather than via the slower, routed Internet. We show that the FFPF framework, with the FPL language and the transmit capabilities discussed in Chapter 5 is powerful enough to be deployed in this application domain also.

8.1 Introduction

Grid and other high-performance applications tend to require high bandwidth end-to-end connections between grid nodes. Often the requirements are for several gigabits per second. When spanning multiple domains, fibre optic networks owners must cooperate in a coordinated manner in order to provide high-speed end-to-end optical connections. Currently, the administrators of such connections use paper-based long-term contracts. There exists a demand for a mechanism that dynamically creates these fast end-to-end connections (termed *lightpaths*) on behalf of grid applications. The use of lightpaths is also envisaged for applications that are connected through hybrid networks [101, 102]. A hybrid network contains routers and switches that accept and forward traffic at layers 1, 2, or 3. In other words, hybrid networks consist of traditional (layer 3) routed networks which allow for optical (layer 1 or 2) shortcuts for certain parts of the end-to-end path. Currently, the peering points between routed networks of the Internet Service Providers (ISPs) by way of the Border Gateway Protocol (BGP) policies determine statically what traffic bypasses the (slow) routed transit net-

work and which links they will use to do so. However, when considering hybrid networks interconnected over long distances, we would like the peering points to play a more active/-dynamic role in determining which traffic should travel over which links, especially since multiple links often exist in parallel. Therefore, an important role for the peering points is path selection and admission control for the links.

Figure 8.1 shows a hybrid network composed of three ISPs (ISP_A, ISP_B, and ISP_C) interconnected both through routed networks (regular Internet) and also through two different optical links managed by different owners. The connections across the optical links is via the peering points which we will call PP_A, PP_B, and PP_C, respectively. An example of such a connection may be a high-bandwidth transatlantic link. We stress that ISP_A and ISP_C may be full-blown ISPs offering extra services to their clients, but they could also be small local ISPs, organisations, departments, research groups, etc.

Users X and Y on the left access servers on the right. We want them to bypass the routed Internet and use optical shortcuts instead. However, while not precluded by the model, we do not require each user to have an individual relation with each ISP and shortcut along the path. Indeed, the link owners should not normally have a notion of which specific IP addresses are allowed to access the link. Instead, we expect ISP A (say, the organisation, department or research group) to have a contract with the link owners and decide locally (and possibly at short timescales) who should get access to the links in domains that are not under its control. For scalability, the model allows the system to be federated along more levels, where ISP_A contacts the authorisation at the next level, which in turn contacts a server at a higher level still, etc. In practice, the number of levels is expected to be small (often one).

A flexible way of bypassing the routed Internet is to have ISP_A tag traffic from X and Y with some sort of token to signal to remote domains that they should be pushed across the optical shortcuts. At the same time, we want to prevent rogue users (e.g., the client of ISP_B indicated in the figure) to tag their packets with similar tokens to gain unauthorised access to the shortcuts. So we need a good user-network interface.

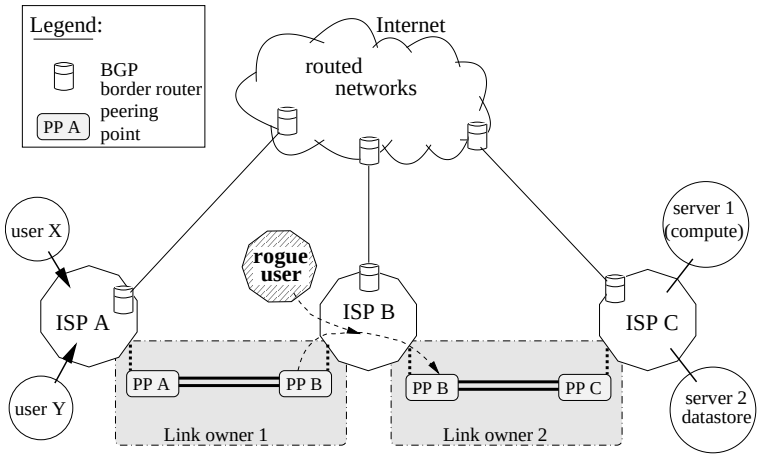


Figure 8.1: Peering in hybrid networks: end-to-end splicing of optical paths.

In principle, the signalling of peering requests to the peering points PP_A, PP_B, and PP_C

can be done either *out-of-band*, or *in-band*. In *out-of-band* signalling there is an explicit control channel to PP_A and PP_B, separate from the data channel(s). In an in-band mechanism, the admission control is based on specific data inserted into the communication channel.

For obtaining authorisation, this case study favours the *push* model as described in the *Authorisation Authentication Accounting* (AAA) framework (RFC 2904) [103]. In the push model, the client (e.g., ISP_A in the figure) contacts an AAA server separate from the datapath to obtain authorisation for the use of network resources (for instance, the optical shortcuts). When the client is ready to use the resources, the client pushes the proof of such authorisation to the service equipment (e.g., network devices). The authorisation is then checked and access is granted or rejected depending on the outcome. An advantage of the push model is that time of authorisation is decoupled from time of use.

The push model of authorisation is compatible with either form of signalling. In the token based switch we opt for *in-band* signalling for reasons of flexibility resulting from the per-packet granularity. Specifically, we insert tokens into each packet as proof of authorisation. Tokens are a simple way to authorise resource usage which may convey different semantics. For instance, we may specify that only packets with the appropriate token are allowed to use a pre-established network connection in a specific time-frame and embed these tokens in the packets of an application distributed over many IP addresses.

However, our tokens differ from common flow identifiers (e.g., ATM VPI/VCI pairs, MPLS labels, and IPv6 flow labels) in that they cannot be tampered with and that they may be associated with arbitrary IP addresses. In essence, tokens bind packet attributes to an issuing attribute authority (e.g., an AAA server in our case). Tokens could be acquired and subsequently used anonymously. Moreover, a token can bind to different semantics (e.g., a user, a group of users, or an institute) and decouples time of authorisation from time of use. In the switch described in this chapter, tokens are used to select shortcuts and different tokens may cause packets to be switched on different links, but in principle they could also be used to enforce other QoS-like properties, such as loss priorities.

In this chapter we describe the Token Based Switch which explores an extreme in the design space of hybrid networks, in which link authorisation/authentication occurs on a per-packet basis in a switch. In addition, each packet is authenticated individually by means of cryptographically signed tokens. The token is used in the switching process to determine on which output port the packet should be forwarded. We realise that this is a radical solution, as encryption and decryption are known to be computationally expensive and most of the overhead is incurred for each and every packet. In practice, however, we found that by careful engineering we are able to sustain multi-gigabit link rates even with a processor that is already four years old at the time of writing, while increasing latency by approximately 15 microseconds per shortcut.

Per-packet authorisation (as first described in [104]) allows shortcuts in third-party networks to be used in a safe manner without having to specify in advance which packets may use the link. Rather, the user has an agreement with the link owner and determines itself which packets from its clients qualify. A user in our context will often not be an individual end user (although this is not precluded by the model), but perhaps an organisation or department. At the same time the authorisation makes sure that no unauthorised 'rogue' packets use the link. The solution allows for novel ways of managing critical network resources.

This chapter describes both the design of the Token Based Switch (TBS) and its imple-

mentation on high-speed network processors. TBS introduces a novel and rather extreme approach to packet switching for handling high-speed link admission control to optical short-cuts. The main goal of this project was to demonstrate that the TBS is feasible at multi-gigabit link rates. In addition it has the following goals: (1) path selection with in-band admission control (specific tokens gives access to shortcut links), (2) external control for negotiating access conditions (e.g., to determine which tokens give access to which links), and (3) secured access control.

The third goal is also important because part of the end-to-end connection may consist of networks with policy constraints such as those meant for the research application domain in the *LambdaGrid*). Moreover, a critical infrastructure needs protection from malicious use of identifiers (e.g., labels in (G)MPLS, header fields in IPv4, or flowID in IPv6). For these reasons, our token recognition uses cryptographic functions, for example, to implement the Hash function based Message Authentication Code (HMAC) (see RFC2104 [105]). An external control interface is required to negotiate the conditions that give access to a specific link. Once an agreement has been reached, the control interface should accept an authorisation key and its service parameters that will allow packets to access the owner's link. To operate at link speeds we push all complex processing to the hardware. In our case we use a dual Intel IXP2850 with on-board crypto and hashing units.

Several projects address the issue of setting up lightpaths dynamically (e.g., UCLP [106], DWDM-RAM [107]), and others look at authorisation (e.g., IP Easy-pass [108]). However, to our knowledge, no solutions exist that support both admission control and path selection for high speed network links in an in-band fashion for setting up safe, per-packet-authenticated, optical short-cuts. In addition, our approach makes it possible to set up multiple shortcuts on behalf of applications that span multiple domains. As a result, a multi-domain end-to-end connection can be transparently improved in terms of speed and number of hops by introducing shortcuts.

8.2 Architecture

At present, many techniques can be used to build end-to-end network connections with some service guarantees. For instance, Differentiated Service (DiffServ) [109] manages the network bandwidth by creating per hop behaviors inside layer-3 routers, while Multi Protocol Label Switching (MPLS) [110] establishes a path using label switched routers. However, a more radical approach that is typically adopted for layer-1 switches, uses the concept of a *lightpath* [111, 112]. In this chapter, a *lightpath* is defined as an optical uni-directional point-to-point connection with effective guaranteed bandwidth. It is suitable for very high data transfer demands such as those found in scientific Grid applications [113]. In order to set up a lightpath, a control plane separate from the data forwarding plane is used on behalf of such applications. For multi-domain control, a mechanism is needed that respects local policies and manages the network domains to set up the end-to-end connection (usually composed of multiple lightpaths) on demand.

8.2.1 High-level overview

To provide higher-level functionality such as trust, authorisation, we layer a service plane above legacy networks [114]. The service plane bridges domains, establishes trust, and exposes control to credited clients/applications while preventing unauthorised access and resource theft. An important component of the service plane is the *authentication, authorisation, and accounting* (AAA) subsystem. The AAA server is the entity that receives a request for resource usage and makes a policy based authorisation decision using information contained within the request, information concerning the resource and possibly information from the stakeholders. After an authorisation decision has been made it replies to the entity that sent the request. The AAA server is considered the single authority governing access to the underlying service equipment. In the next subsections, we explain the parts of the service plane that pertain to the token based switch.

Push-based authorisation and in-band signalling

As explained in RFC 2904, various models of interaction for requesting an authorisation and subsequently using the authorisation for gaining access to resources are possible between AAA clients, AAA server, and the service equipment representing the network resources (like optical shortcuts). For instance, to request an authorisation the clients can contact either the AAA Server or the service equipment. In the latter case the service equipment in turn outsources the access decision to the AAA server. Access to resources is subsequently enforced by the service equipment on the basis of the AAA server's decision. This is known as a *pull* model. Alternatively, the AAA client may send a request to the AAA server directly and receive the AAA server's decision in the form of a (secure) token. The user must then push this token to the service equipment to prove authorisation. We refer to this interaction as the *push* model.

As explained earlier, our architecture uses in-band signalling and push-based authorisation. Compared to out-of-band signalling, the in-band model on average incurs more overhead but less state at the service equipment. An advantage of push-based authorisation is that it allows temporal separation of the process that obtains an authorisation (which can be a very complex process) and the process that enforces the authorisation. As mentioned earlier, tokens can be acquired and subsequently used anonymously, and may bind to different semantics (e.g., a user, a group of users or an institute). We implement the most radical form of push-based in-band signalling in which every packet is authenticated individually.

Obtaining and using authorisation

For an application to receive an end-to-end connection with pre-allocated bandwidth, the in-band push model dictates that a token must be inserted in its network data. In order to obtain tokens, the user or a user organisation may contact the ISP's local AAA server with the appropriate request. Normally, users deal only with their local ISP which may be a commercial ISP or an entity as small as a department, faculty, organisation, etc. The ISP may determine independently which traffic should use the shortcuts. The ISP is assumed to have previously obtained authorisation to use optical shortcuts from the operators of optical links using similar principles as those used between the client and its local ISP. Indeed, there may

be a chain of AAA servers involved in the full authorisation process. Once the authorisation has been obtained, the ISP can use it as it sees fit. Figure 8.2 shows the architecture used for setting up a *lightpath* using token based networking.

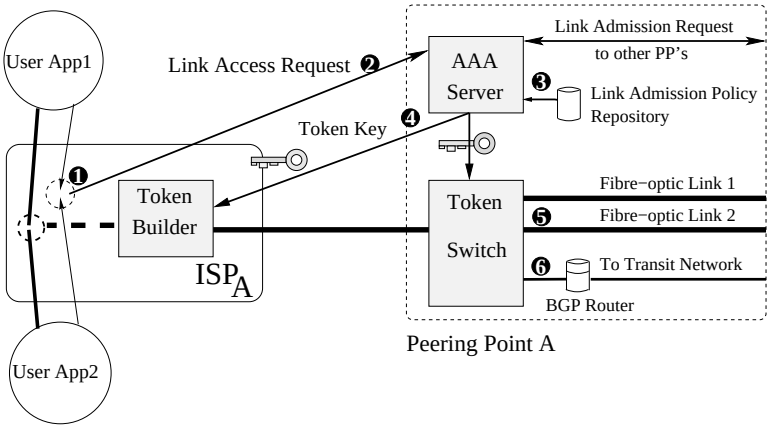


Figure 8.2: Token based networking architecture.

In a nutshell, the process is as follows. On behalf of a subset of its users, ISP_A generates a Link Access Request (LAR) message to use a particular link (from the available links and its peering point) for a certain period of time ①. The details of LAR generation based on user requests are beyond the scope of this chapter. Interested readers are referred to [114]. The LAR is sent to the AAA server of peering point A ②. The AAA server fetches a policy from the policy repository that validates the request ③.

Next, a key (*TokenKey*) is generated and sent to ISP_A and peering point A ④. ISP_A may use the key to create a token for each packet that should use the link. The token will be injected in the packets which are then forwarded to the peering point. The entity responsible for token injection is known as the token builder. On the other side, peering point A uses the same *TokenKey* as ISP_A to check the incoming token-annotated packets. In other words, for each received packet, peering point A creates a local token and checks it against the embedded packet token. The packet is authenticated when both tokens match. An authenticated packet is then forwarded to its corresponding fibre-optic link ⑤. All other packets are transmitted on a 'default' link that is connected to the routed transit network ⑥. The entity responsible for token checking and packet switching is known as the token switch. We show later that we currently use an IP option field to carry the tokens, but other implementations are possible. The option field is useful for backward compatibility as it is supported in IPv4 and will be ignored by normal routers.

When a packet arrives at the token switch, we must find the appropriate key to use for generating the token locally. Which key to use is identified by fields in the packet itself. For instance, we may associate a key with an IP source and destination pair, so all traffic between two machines are handled with the same key. However, other forms of aggregation are possible. For instance, we may handle all traffic between multiple hosts running a common application such as machines participating in a Grid experiment, etc. In general, we allow the key to be selected by means of an *aggregation identifier*, embedded in the packet. The

aggregation identifier is inserted in the packet together with the token by the ISP to signal the key to use.

Summarising, the TBS architecture offers to the user applications an authenticated access control mechanism to critical high-speed links (lightpaths) across multi-domain hybrid networks. The procedure consists of two phases that are decoupled in time: (1) a high-level set-up phase (obtaining tokens from an AAA web-service), and (2) a fast datapath consisting of low-level authorisation checks (per-packet token checks at network edges within a multi-domain end-to-end connection). In other words, the first phase allows individual users, or group of users (e.g., a research institution), or even user applications, to request privileged end-to-end connection across multi-domain networks by contacting only one authority: their own ISP. The second phase determines how TBS authenticates network traffic (TCP connections, UDP transmissions, or other protocols) and how it checks the traffic for authorisation on behalf of their applications. The second phase is also responsible for preventing malicious use of lightpaths in a multi-domain network. Two network components are involved in the datapath: the token builder and the token based switch.

8.2.2 Token principles

Compared to other mechanisms (such as certificates), a token is a general type of trusted and cryptographically protected proof of authorisation with flexible usage policies. A token created for an IP packet is essentially the result of applying an HMAC algorithm over a number of packet fields as illustrated in Figure 8.3 and explained below.

An HMAC algorithm is a key-dependent way to create a one-way hash. In order to ensure the token uniqueness for packets, we should try to include fields that are different in every packet. Therefore, a part of the packet data will be included together with the packet header in the HMAC calculation. On the other hand, we must exclude information that changes at each hop such as IP's time to live (TTL) field.

An HMAC algorithm proposed to create such a one-way hash (token) is HMAC-SHA1 (RFC 2104). In our implementation we opted for a strong proof of authorisation by means of HMAC-SHA1. It may be possible to use more lightweight algorithms such as RC5 which is also used by IP EasyPass [108]. However, we wanted to evaluate the performance that could be achieved with strong authentication. If anything that using RC5 or similar algorithms would only make it scale better.

The HMAC-SHA1 algorithm needs a 20 bytes key to encrypt data blocks in chunks of 64 bytes each and provides a unique result of 20 bytes. Therefore, we built the system such that (1) the AAA server provides a *TokenKey* of 20 bytes as encryption key, and (ii) we use this key to encrypt the first 64 bytes of packet data after masking out all information that is to be excluded. After all, as the first 64 bytes cover most important packet fields (e.g., IP headers) for the entire packet, they are well-suited for providing the necessary security guarantees.

To evaluate the TBS principles, we developed a prototype that stores the token in each packet's IP option field (as defined in RFC 791). An IP option can be of variable length and its field will be ignored by normal routers. In other words, parts of the path may consist of routers and parts of the path may be established using fast switches. Although some routers have a slower processing path for the IP option packets than simple IP packets (because higher level headers will be at different positions in the packet), we noticed that our TBS system

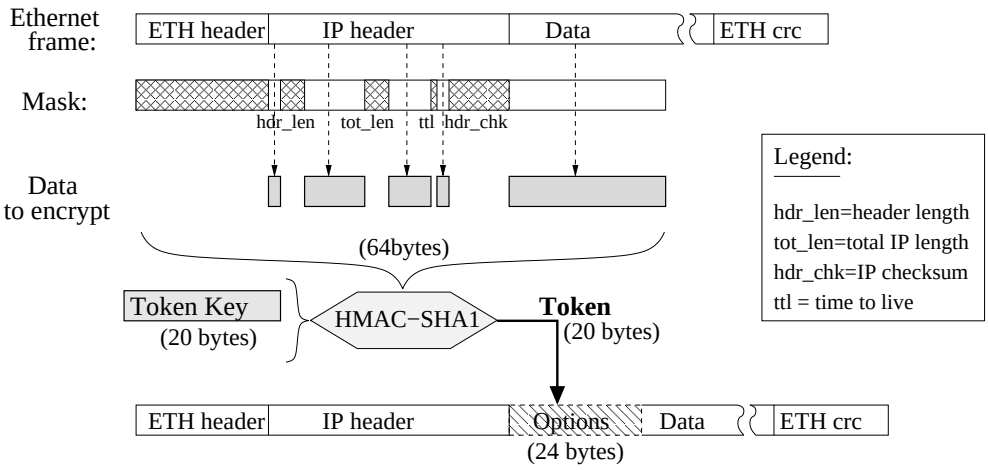


Figure 8.3: Token creation.

works well in high speed, important and pricey sites (e.g., ISPs, grid nodes interconnection points) where all systems and also routers are updated to the state-of-the-art hardware. This is exactly the environment to which we want to cater. We stress, however, that the IP option implementation is for prototyping purposes. More elegant implementations may use a form of MPLS-like shim headers.

Figure 8.3 shows the process of token creation and insertion in the IP option field. In our prototype, the HMAC-SHA1 algorithm generates the unique token (20 bytes) that will be injected into the packet's IP option field. As an IP option field has a header of two bytes, and network hardware systems commonly work most efficiently on chunks of four or eight bytes, we reserve 24 bytes for the IP option field. In other words, we have two bytes available for the aggregation identifier which enables us to distinguish 64K aggregates and seems sufficient for now. If more are needed, we may consider adding more bytes to the option field. Alternatively, one may use labels in existing protocols as aggregation identifier (e.g., the IPv6 flow identifiers).

We mention again that some of the first 64 bytes of an Ethernet frame are masked in order to make the token independent of the IP header fields which change when a token is inserted (e.g., header length, total length, IP checksum) or when the packet crosses intermediate routers (e.g., TTL). The mask also provides flexibility for packet authentication, so that one could use the (sub)network instead of end node addresses, or limit the token coverage to the destination IP address only (e.g., by selectively masking the IP source address).

8.3 Implementation details

As mentioned earlier, the Token Based Switch (TBS) performs secure lightpath selection on-the-fly on a per-packet basis by means of cryptographically-protected authentication. Therefore, token generation and verification at high speeds is crucial. As the latest generation of Intel's IXP Network Processor Units (NPs) includes crypto units in hardware, it provides an

ideal platform for TBS' packet authentication. The IXP28xx family of network processors was introduced in 2002 for high-end networking equipment and, while six years old technology, it comes equipped with hardware assists that make it an ideal fit for TBS.

On the other hand, NPs are architecturally complex and difficult to program. For instance, programmers have to deal with multiple RISC cores with various types of memories (both local and shared), small instruction stores, and a single XScale-based control processor. Therefore, building software applications like the TBS on NPs from scratch is a challenging task. However, in previous work we have shown how programming can be eased by means of a robust software framework and special-purpose programming languages [14, 77].

In summary, we use the Intel IXDP2850 dual NP as our hardware platform [48], and extended the implementation of the Fairly Fast Packet Filter (FFPF) on NPs as our software framework [14]. In particular, we added specific IXDP2850 features like hardware supported encryption and dual NPs. As the Intel IXDP2850 kit is expensive network equipment we decided for prototyping purposes to implement both the token builder and the token switch on the same machine, running the builder on one NP and the switch on the other. While in reality the two functionalities would be separated, the setup makes little difference from a prototype point of view (except that performance is roughly half of what we could achieve if we used both NPs for one purpose). In the next few sections, we describe our implementation in more detail.

8.3.1 Hardware platform

The main reason for opting for the IXP2850 NP is that it provides high speed packet handling (up to 10 Gbps) and on-chip crypto hardware. Figure 8.4 shows the hardware architecture used in the implementation of the token switch. Our prototype uses the IXDP2850 development platform, consisting of dual IXP2850 NPs ① & ②, 10×1 Gbps fibre interfaces ③, a loopback fabric interface ④ and fast data buses (SPI, CSIX). Each NP has several external memories (SRAM, DRAM) and its own PCI bus for the control plane (in our setup it connects to a slow 100 Mbps NIC). In addition, each 2850 NP contains on-chip 16 multi-threaded RISC processors (μEngines) running at 1.4GHz, a fast local memory, lots of registers and two hardware crypto units for encryption/decryption of commonly used algorithms (e.g., 3DES, AES, SHA-1, HMAC). The μEngines are highly-specialised processors designed for packet processing, each running independently from the others from private instruction stores of 8K instructions. See Chapter 2, Section 2.2.4 for more details.

As illustrated in Figure 8.4, the incoming packets are received by the Ingress NP (via the SPI bus). These packets can be processed in parallel with the help of μEngines. The packets are subsequently forwarded to the second NP via the CSIX bus. The second NP can process these packets and then decide which will be forwarded out of the box (via the SPI bus) and which outgoing link will be used.

8.3.2 Software framework: FFPF on IXP2850

For clarity, let us briefly recapitulate the key characteristics of Fairly Fast Packet Filter (FFPF). The FFPF is a flexible software framework designed for high-speed packet processing. FFPF supports both commodity PCs and IXP Network Processors natively and has a

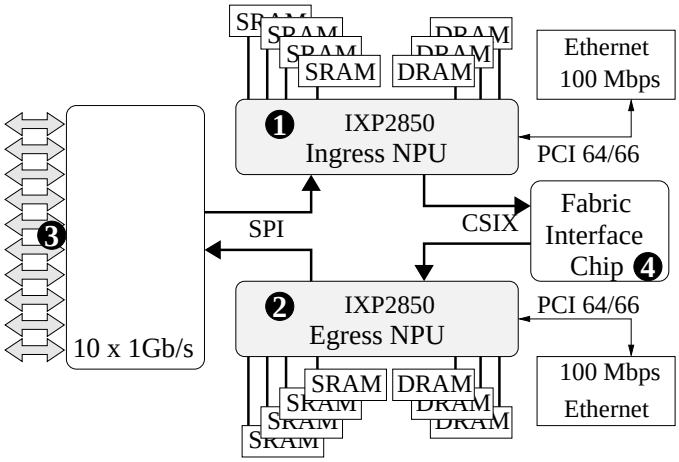


Figure 8.4: IXDP2850 development platform.

highly modular design. FFPF was designed to meet the following challenges. First, it exploits the parallelism offered by multiple cores (e.g., the host CPU and the IXP's μEngines). Second, it separates data from control, keeping the fast-path as efficient as possible. Third, FFPF avoids packet copies by way of a 'smart' buffer management system. Fourth, the FFPF framework allows building and linking custom packet processing tasks inside the low-level hardware (e.g., the μEngines of each NP).

For example, a packet processing application may be built by using the FFPF framework as follows. First, the application is written in the FPL packet processing language introduced in Section 3.1, and compiled by the FPL-compiler. Second, the application's object code is linked with the FFPF framework. Third, the code is loaded into the hardware with the help of the FFPF management tools, as explained in Section 4.2. Most of the complexity of programming low-level hardware (e.g., packet reception and transmission, memory access, etc.) is hidden behind a friendly programming language.

As mentioned earlier, in the prototype one of the NPs on the IXDP2850 is used for the implementation of the token builder and the other one for the token switch. In other words, the token builder and token switch are physically co-located, while logically they are separate. In reality the token builder and switch may share the same PoP, but they will not normally be in the same machine. Note that such a configuration makes no difference for the performance measurements, except that we only use half of our processor for each side.

As illustrated in Figure 8.5, the FFPF implementation on the IXDP2850 consists of the following software modules: ① `Rx` (for receiving packets from gigabit ports), ② $\texttt{Tx}_{csix}$ (for forwarding packets to the second NP via the CSIX bus), ③ $\texttt{Rx}_{csix}$ (for receiving packets from the first NP), ④ `Tx` (for transmitting packets to the gigabit ports), ⑤ hardware supported extensions for crypto algorithms (e.g., 3DES, AES, SHA1, HMAC) and ⑥ the Buffer Management System. The buffer management system consists of a single shared packet buffer (*PBuf*), one index buffer (*IBuf*) for each processing task and one transmit buffer (*TBuf*) per NP. All remaining μEngines are available for packet processing tasks. In our case, we will use them for token building and switching.

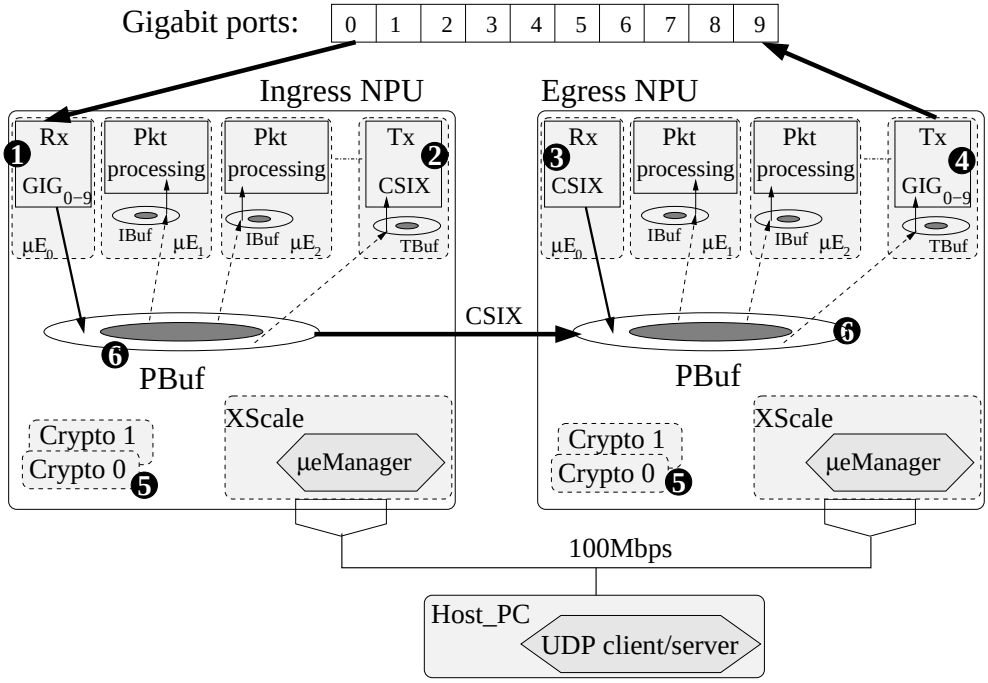


Figure 8.5: FFPF implementation on dual IXP2850.

The FFPF implementation on the IXDP2850 works as follows. When a packet arrives on a gigabit port, the first μEngine of the Ingress NP (`Rx`) pushes it to the shared buffer *PBuf* and updates the write index. The packet processing μEngines (indicated as *pkt processing* in the figure) detect when a new packet is available by checking their local read index against the global write index and starts processing it directly from *PBuf*. As a result, the current packet can be pushed to other dependent processing tasks (by pushing its index to the task's local buffer *IBuf*), it can be forwarded out to the Egress NP (by pushing its index in the transmit buffer (*TBuf*), or it can be dropped. An *IBuf* essentially provides a *view* on the (shared) incoming stream and may differ from consumer to consumer.

IBuf helps multiple consumers to share packet buffers without copying. In addition, the indices in the *IBuf*s may contain integer classification results. For instance, if a classifier function classifies TCP/IP packets according to their destination port, subsequent functions that process the packets find the port number immediately, i.e., without having to retrieve the packets from memory. This is useful, as memory access is slow compared to processor speeds. Moreover, *IBuf* are stored in SRAM, which is faster, but smaller than the DRAM that is used to store the packets.

The last μEngine of the Ingress NP (`Tx`) takes each packet scheduled for transmission (by its *TBuf*) and sends it to the Egress NP through a fast local bus (CSIX). Similarly, on the Egress NP, the data processing works in the same way as in the Ingress NP except that it receives packets from the internal CSIX bus and sends them out of the box through a specific gigabit port.

State-of-the-art packet processing systems must handle (receive, process and transmit) traffic data at very high speeds (gigabits/sec). Therefore, application design is commonly split in two parts: a control path (control applications using a slow management link), and a data path (packet processing applications using gigabit links).

Control path

The control path in the FFPF implementation on the IXDP2850 consists of a client/server connection between a host PC (an x86 running Linux) and each NP control processor (an XScale running embedded Linux). The system is remotely controlled by a user tool running on the host. The FFPF framework provides these tools in its 'management' toolkit. An FFPF μeManager that is running on each NP XScale establishes a connection to the host PC and is responsible for managing its hardware components as follows: (1) it initialises the internal chipset (e.g., fibre media, fabric chips), (2) it synchronises the internal buses (e.g., SPI, CSIX), (3) it initialises the μEngines and memory (e.g., buffers), (4) it loads/unloads code on/from μEngines, (5) it starts/stops requested μEngines and (6) it updates a local shared memory placed in SRAM and known as `KeysTable` on request by the host PC. Each entry in the `KeysTable` contains a key *K* and an authorised port *Np* for each aggregation identifier. As we do not have a large scale deployment and statespace is not a problem, in the evaluation our prototype implementation simply uses the source and destination IP address pair as aggregation identifier.

Data path

The data path is composed of all processing performed for the Token Based Switch on incoming packets. As mentioned earlier, the FFPF framework takes care of basic processing steps like receiving of packets, storing them in a shared buffer, making them available to processing tasks and packet transmission.

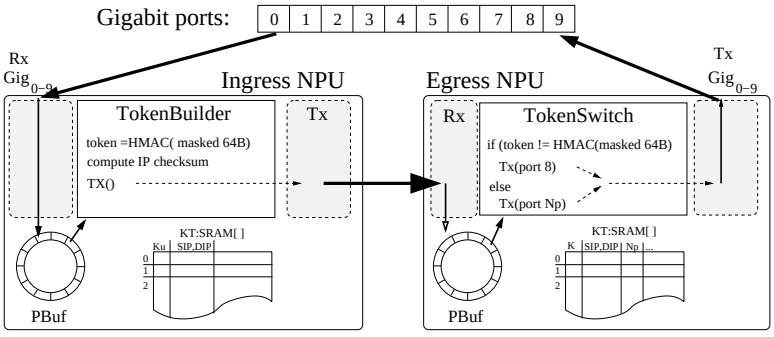


Figure 8.6: Token Based Switch using a dual IXP2850.

```
1  MEM:SRAM[2];
2  MEM[64];
3  REG[20];
4  MEMCPY(M[0], PKT.IP_HDR && MEMS[4], 64); // mask the IP header and store it
       in a LocalMem array
5  R[0] = EXTERN(Crypto, HMAC-SHA1, MEMS[0], M[0]); // where MEMS[0] is the
       key from SRAM table and M[0] is the source field (masked packet)
6  PKT.B[14].LO += 6; // 6 blocks of 4Bytes more for IPoption field into
       IP_HDR_length
7  PKT.W[8] += 24; // 24 more bytes for IPoption field into Total_IP length
8  PKT.W[17] = 0x8E18; // IPoption_type as: Hdr(2bytes): Type:Copy + Control +
       Experimental_Access (1Byte) + IP_option_Length (1Byte) + Token
       (20bytes)
9  MEMCPY(PKT.W[18],R[0], 20); // copy the encrypted token from R[0] into the
       packet's IPoption field
10 EXTERN(Checksum, IP); // re-compute the IP checksum
11 TX();
```

Listing 8.1: The Token Builder application written in FPL.

8.3.3 Token Based Switch

The TBS application consists of two distinct software modules: the token builder and the token switch (see also in Figure 8.2). In our prototype, the application modules are implemented on the same hardware development system (IXDP2850) although in reality they are likely to be situated in different locations. Therefore, our implementation consists of a demo system as shown in Figure 8.6.

The token builder application module is implemented on two μEngines in the Ingress NP, while the token switch module is implemented on two μEngines in the Egress NP. Although the mapping can be easily scaled up to more μEngines, we use only two μEngines per application module because they provide sufficient performance already, as we will see in Section 8.4, the bottleneck is the limited number of crypto units.

The **token builder** application implements the token principles as described in Figure 8.3. The FFPF software framework automatically feeds the token builder module with packet handles. As described in Listing 8.1, the token builder retrieves the first 64 bytes of the current packet from the shared *PBuf* memory into local registers and then applies a custom mask over these bytes in order to hide unused fields like IP header length, IP total length, etc. The application also retrieves a proper token key (`K`) from a local KeysTable by looking up the aggregation identifier (e.g., a flow identified by the IP source address and/or IP destination address pair, or other aggregates). Next, an HMAC-SHA1 algorithm is issued over these 64 bytes of data using `K` (20 bytes) as encryption key. The encryption result (20 bytes) is then 'inserted' into the current packet's IP option field. This operation involves shifting packet data to make space for the option field. It also involves re-computing its IP checksum because of the IP header modification. Once the packet has been modified it is scheduled for transmission. In this prototype, the ingress NP packets are transmitted out to the egress NP via a fast bus.

The **token switch** application implements the token switch machine from the system architecture (see Figure 8.2 in Section 8.2). The FFPF software framework automatically

```
1  MEM:SRAM[1024];
2  MEM[64];
3  REG[20];
4  MEMCPY(M[0], PKT.IP_HDR && MEMS[4], 64); // mask the IP header and store it
       in a LocalMem array
5  R[0] = Hash(M[0], 24, 0x400); // get an 'unique' index of max. 1024
6  R[1] = 0;
7  FOR (R[2]=0; R[2]<1024; R[2]++) // look for the entry into the KeysTable
8    IF (MEMS[R[2]] == R[0]) THEN R[1]=1; BREAK; FI;
9  ROF;
10 IF (R[1]) THEN // if an authorized entry found in KeysTable for this pkt.
11   R[3] = EXTERN(Crypto, HMAC-SHA1, MEMS[R[2]].K, M[0]); // where .K is the
         key from SRAM KeysTable and M[0] is the source field (masked packet)
12   IF (MEMCMP(R[3], PKT.W[18], 20) == 0)  // if the tokens match
13     THEN TX(MEMS[R[2]].AuthPort); // send it to the authorised port
14     ELSE TX(9);                   // send it to the default port 9
15   FI;
16 ELSE
17   TX(9);
18 FI;
```

Listing 8.2: The Token Switch application written in FPL.

feeds the token switch module with packet handles. The token switch checks whether the current packet has the appropriate IP option field, and extracts the first 64 bytes of the original packet data and the token key value (`y'`) from the option field into local registers. Next, it applies a custom mask over the 64 bytes of data. As illustrated in Listing 8.2, the application also retrieves a proper token key (`K`) from its local KeysTable by looking up the aggregation identifier (for instance, the IP source address and/or destination address pair). If no entry is found in the KeysTable the packet cannot be authorised and it will be sent out to a default port (e.g., port 8) for transmission over a (slow) routed connection. Otherwise, an HMAC-SHA1 algorithm is issued over the 64 bytes of data and using the token key value (20 bytes) as encryption key. The encryption result (`y"`) is compared to the built-in packet token (`y'`). When they match, the packet has been successfully authorised and it will be forwarded to its authorised port (`Np`).

8.4 Evaluation

Figure 8.7 shows the system setup used for proving the concepts of token based switching [104].

The IXDP2850 development system has two IXP2850 NPs (Ingress and Egress) that boot from a Linux boot server machine. At runtime we use the FFPF software framework for the control path (running on the Linux server and both embedded Linux NPs). Three other Linux machines (rembrandt 6, 7 and 8) serve as clients and are each connected via gigabit fibre to an NP gigabit port. In order to find out the maximum data rate the proposed system can handle, we evaluate the following scenario:

- An UDP traffic stream was generated (using `iperf` tool [115]) from `Rembrandt6`

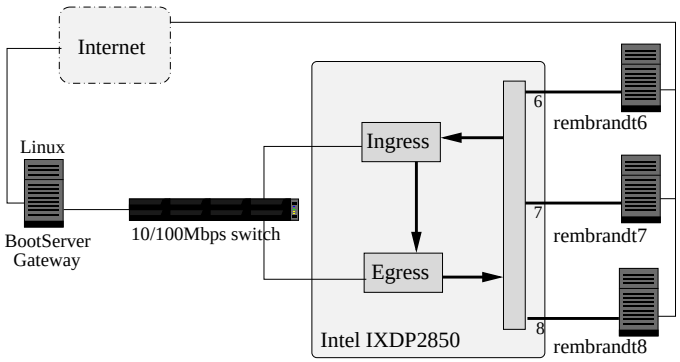


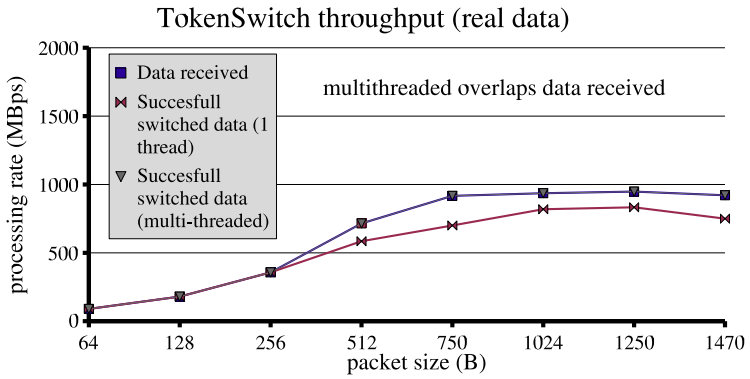
Figure 8.7: Token Based Switch demo.

to `Rembrandt7`;

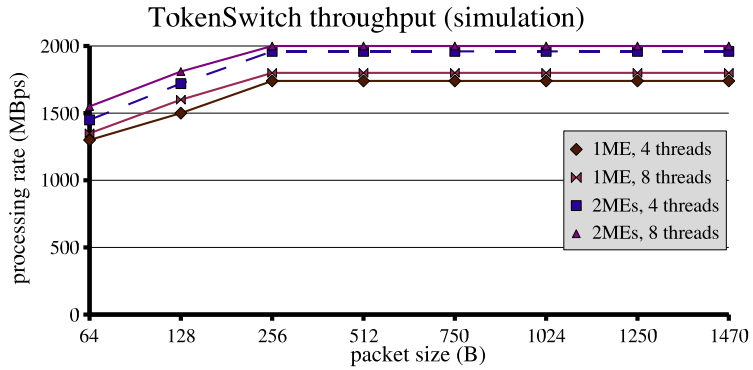
- A token key was set for authorising traffic between (Rembrandt6 - Rembrandt7) to end up on port 7 and another token key was set for authorising traffic from (Rembrandt7 - Rembrandt6) to end up on port 6;
- Unauthorised traffic should go to the default port (e.g., port 8):
- To prove that authorised traffic ends up on port 7, Rembrandt7 was connected to the IXDP2850 port 7 and `tcpdump` was listening to the gigabit port;
- To prove that unauthorised traffic ends up on port 8, Rembrandt8 was connected to the IXDP2850 port 8 and `tcpdump` was listening to the gigabit port.

The performance of the above described test is shown in Figure 8.8.a. It has two charts: (1) 'data received' which represents the received rate in the IXDP2850 box and (2) 'successfully switched data' which denotes the rate that the TBS could handle properly using just a single thread for processing. The 'data received' chart is low for small packets because of the Gigabit PCI card limitation used in the Rembrandt6 PC for traffic generation. So, for small packets it reflects the limitations of the traffic generator rather than those of the TBS. The second chart, 'Successfully switched data', is lower than the first one for high speeds because we are using a single threaded implementation. The multi-threaded version coincides exactly with the 'data received' chart and is thefore not visible.

While we cannot predict the real performance for speeds above 1 Gbps without performing measurements with a high-speed traffic generator, we estimated the outcome by using the Intel's cycle accurate IXP simulator running in debug mode. Table 8.1 and Table 8.2 show the cycle estimation for a 150 bytes packet processed by each software component from the Data path (Rx_Gig, Tx_Gig, Rx_CSIX, Tx_CSIX, TokenBuilder and TokenSwitch). Table 8.1 illustrates the cycles spent for one packet in each software module of the FFPF implementation on IXP2850. These modules are optimised for multi-threading packet processing (e.g., receiving, storing, transmitting). The first row in Table 8.2 contains the cycles spent for one packet in a single threaded version of the token builder and token switch modules. We note that these values are high because all subtasks (e.g., encryption, token insertion, checksum



(a) running in hardware



(b) running in cycle accurate simulator

Figure 8.8: Token Based Switch performances

computation) run linearly (no parallelism involved at all) and use only one crypto unit each. This single threaded version gives the performance shown in Figure 8.8.a. The next rows illustrate various implementations of the multi-threading version. Normally, we should expect better performance when we increase parallelism (e.g., more threads, or more μEngines) because of real parallelism and because memory latencies are hidden with multiple threads. On the other hand, one may also expect more memory contention and synchronisation overhead, so without real measurements it is impossible to draw accurate conclusions about the performance. However, having only two crypto units available per NP limits the performance to the value of roughly 2000 cycles (the token switch module spends its time mostly on encryption, while the token builder module does also token insertion in the packet).

Note that our prototype implements each TBS module (token builder and token switch) on only one NP of the IXDP2850 hardware system. The reason this was done is that we have only one of these (expensive) IXDP2850 devices available in our lab. In a real setup, however, each TBS module may use the full dual-NP IXDP2850 for building or checking

tokens and therefore the system performance is expected roughly to double compared to our presented figures, mainly because we would benefit from the availability of four crypto units.

FFPF module	Rx_Gig	Tx_CSIX	Rx_CSIX	Tx_Gig
μEngine cycles	408	276	504	248

Table 8.1: FFPF on IXP2850 overhead

TBS	TokenBuilder	TokenSwitch
single threaded	5777	3522
4 threads, 1 μEngine	3133	2150
8 threads, 1 μEngine	3000	2100
4 threads, 2 μEngines	2600	2100
8 threads, 2 μEngines	2500	2000

Table 8.2: TBS overhead

Using the cycle estimation given in Tables 8.1 and 8.2, we express the throughput as a function of the packet size, number of threads and number of μEngines: rate=f(packet size, threads, μEngines) without taking into account additional contention.

As illustrated in Figure 8.8.b, the estimated throughput for our multi-threaded version goes up with roughly 1 Gbps over the single threaded version (Figure 8.8.a). This improvement is due to the increased parallelism (e.g., more threads and more μEngines) that hides the memory latency and due to the second crypto unit usage.

We also measured the latency introduced by our Token Based Switch system. The token builder application (the whole Ingress NP chain) takes 13.690 cycles meaning a 9,7μs processing time (introduced latency), and the TokenSwitch application (the whole Egress NP chain) takes 8.810 cycles meaning a 6,2μs latency. We mention that a μEngine in the IXP2850 NP runs at 1400MHz.

8.5 Discussion

In addition to commercial solutions for single domain provider-controlled applications such as Nortel DRAC, Alcatel BonD, some research is also underway to explore the concept of user-controlled optical network paths. One of the leading software packages is the User Controlled Lightpath Provisioning (UCLP) [116]. UCLP currently works in a multi-domain fashion, where all parties and rules are pre-determined. Truong et al [117] worked on policy-based admission control for UCLP and implemented fine-grained access control.

Some interesting work in the optical field is also done in Dense Wavelength Division Multiplexing-RAM [118], where a Grid-based optical (dynamic) bandwidth manager is created for a metropolitan area. Our approach is different in the sense that we provide a mechanism to *dynamically* set up *multiple shortcuts* across a multi-domain end-to-end connection. Therefore, an end-to-end connection can be easily improved in terms of speed and hop count by introducing 'shortcuts' based on new user's agreements.

IP Easy-pass [108] proposed a network-edge resource access control mechanism to prevent unauthorised access to reserved network resources at edge devices (e.g., ISP edge-routers). IP packets that are special demanding, such as real-time video streams, get an RC5 encrypted pass appended. Then, at edge-routers, a Linux kernel validates the legitimacy of the incoming IP packets by simply checking their annotated pass. Unlike our work, the solution aims at fairly low link rates. While our solution shares the idea of authentication per packet (token), we use a safer encryption algorithm (HMAC-SHA1) and a separate control path for key management (provided by AAA servers). In addition, we demonstrate that by using network processors we are able to cope with multi-gigabit rates.

Most related to our TBS is Dynamic Resource Allocation in GMPLS Optical Networks (DRAGON) framework [119]. This ongoing work defines a research and experimental framework for high-performance networks required by Grid computing and e-science applications. The DRAGON framework allows dynamic provisioning of multi-domain network resources in order to establish deterministic paths in direct response to end-user requests. DRAGON's control-plane architecture uses GMPLS as basic building block and AAA servers for authentication, authorisation and accounting mechanism. Thereby, we found a role for our TBS within the larger DRAGON framework and we currently work together to bring the latest TBS achievements into DRAGON.

8.5.1 Summary

This chapter presented our implementation of the Token Based Switch application on Intel IXP2850 network processors, which allows one to select an optical path in hybrid networks. The admission control process is based on token principles. A token represents the right to use a pre-established network connection in a specific time frame. Tokens allow separation of the (slow) authorisation process and the real-time usage of high-speed optical network links. The experimental results suggest that a TokenSwitch implementation using the latest Network Processor generation can perform packets authorisation at multi-gigabit speeds.

Chapter 9

Conclusions

The explosion of the Internet in size and amount of traffic makes the administration and services harder and harder to manage. Moreover, new application demands to cope with the growth of some of the Internet problems (intrusions, viruses, spam, etc.) require much more processing per byte of traffic than was needed previously. Earlier, most monitoring tasks may have been enough to use a commodity PC with regular network cards. Currently, even using specialised hardware (e.g., network processors) we cannot build one stand-alone, single-CPU system to support the rising application demands (intrusion detection, virus-scanner, etc.) at multi-gigabit speeds.

In this thesis the use of both parallel and distributed systems is proposed for traffic processing at high speeds. The preceding chapters presented a concept and the implementation of a distributed processing system, and a control system designed to keep the distributed traffic processing system stable regardless of environment changes (e.g., traffic peaks).

The research was started with the following major goal: finding a novel traffic processing architecture that suits the current requirements and scales to future demands. We identified the following requirements:

1. Building a traffic processing system for intensive applications (e.g., IDS, IPS) that supports gigabit link rates requires *parallel* processing so as to cope with the gap between the link and memory speeds as shown in Section 1.4. For higher link rates, we need to *distribute* the workload on a parallel and distributed architecture as introduced in the Chapters 3 and 5, respectively.

2. In order to build a traffic processing system that is less expensive than a supercomputer, has a long life-cycle, and supports a large variety of applications, we need to make use of *heterogeneous* processing nodes in a distributed system. For instance, we can use different network processor generations together with old and new commodity PCs.

3. Writing traffic processing applications on parallel and distributed environment is a difficult task. However, using a high level programming language and compilers for various hardware targets increases the user's productivity and hence, it gives a better time

to market factor. In other words, the language and software framework makes the programmer's life easier by hiding many low-level programming details of often complex parallel hardware (multi-cores). Moreover, the framework allows building applications with fewer bugs related to hardware parallelism or workload distribution.

4. We say that a traffic processing system works in a dynamic environment because the total bandwidth of the incoming traffic and the availability of processing cores vary over time. Thus, the system workload also varies even though it runs the same traffic processing application. When building a robust system, a system that does not fail when it becomes congested for short periods of time, we must use a control mechanism. As our system is very dynamic (processes millions of packets per second), we need an automatic, un-manned, control system to keep the traffic processing stable regardless of environment changes.

9.1 Summary of contributions

This thesis introduced a language, a compiler, and run-time extensions to an existing traffic processing system known as Fairly Fast Packet Filter (FFPF). The run-time extensions were made in respect to support various hardware targets like commodity PCs, network processors, and FPGAs. These extensions pushes FFPF towards heterogeneous distributed traffic processing. Then, the thesis showed a (centralised) control that keeps the distributed heterogeneous traffic processing system stable regardless of the environment changes (traffic variations, hardware failure, or hardware upgrade). In the end, it is used to illustrate as a case study a traffic processing implementation for a specific application domain: per-packet authentication at multi-gigabit speeds.

Chapter 2 presented briefly the FFPF software framework used and extended for the prototypes built during this research. Then, the state-of-the-art hardware – network processors – used in the application development of this research were presented in the end of Chapter 2.

Chapter 3 described the extensions added to the FFPF software framework in order to support the innovative concepts introduced later in the Chapters 5 and 6. First, we designed a new language, FPL, oriented on development of packet processing applications. Then, we designed and developed a compiler that supports multiple hardware targets like commodity PCs, several network processor generations, and a custom FPGA design.

Chapter 4 presented the run-time environments on each supported hardware platform for the FPL compiled applications.

Chapter 5 presented the NET-FFPF distributed network processing environment and its extensions to the FPL programming language, which enable users to process network traffic at high speeds by distributing tasks over a network of commodity and/or special purpose devices such as PCs and network processors. A task is distributed by constructing a processing tree that executes simple tasks such as splitting traffic near the root of the tree while executing more demanding tasks at the less-travelled leaves. Explicit language support in FPL enabled us to efficiently map a program for such a tree.

Chapter 6 described a control architecture for distributed traffic processing systems. The control architecture used a control loop that monitors and adjusts each processing node of the entire distributed system. The stability goal of the control system is achieved by re-mapping

the application tasks from a congested node to another and also by re-distributing the traffic across the distributed processing hierarchy according to the new configuration.

Chapter 7 described an implementation of the centralised adaptive control approach applied to our distributed traffic processing system.

Chapter 8 presented a case study in which the FPL compiler and the extended run-time version of the FFPF framework described in Chapter 3 are applied to perform specific traffic processing at multi-gigabit speeds. The presented application, Token Based Switch, authenticates and routes the incoming traffic according to an encrypted message built-in the packet (a token). This application is a proof of a new networking concept where the data drives the networks. The application makes use of hardware encryption units available in the latest network processor generation, the Intel IXP2850.

9.2 Further research

The proposed concept of traffic processing in a distributed way uses two major idea: (1) traffic splitting, and (2) basic traffic processing. Besides this, a control overlay is needed to keep the entire distributed system stable when the environment changes. However, in the current implementation presented in Chapter 6, some simplified assumption were made. We believe that addressing these assumptions has potential for further improvements.

In our distributed processing environment, as described in Section 5.2, we assumed a tree-like hierarchy. It would be interesting to look at different topologies, especially those which would allow to re-process a stream and hence, the traffic must travel forth and back over the topology.

In Section 7.1.2, our control design used a simple control law inspired by the TCP control flow algorithm. Although we mentioned the advantages of using predictive algorithms, a implementation of such advanced control algorithms may provide a more robust system with a faster response to the environment changes.

Given the current advances in the Storage Area Network (SANs) that support storage over IP, we think that extending our distributed processing system with support for SAN devices would positively affect two important issues: buffering during re-mapping and merging and storing of end-results. Moreover, we could imagine hybrid devices (network processors and SANs) available on the market not so far in the future.

However, there still are open questions in the field of distributed traffic processing systems. For example, when migrating existing compact, single-node applications to a distributed architecture, how can we perform an automatic task partitioning at system deployment stage? The automatic partitioning may efficiently split the main-application into components by means of mapping of specific components on specialised cores. Given several traffic processing applications running on a distributed system composed of many processing nodes, how can we implement an efficient mechanism for retrieving of the processing results?

Bibliography

[1] George Gilder. Fiber Keeps Its Promise: Get ready. Bandwidth will triple each year for the next 25. *Forbes*, April 1997.

[2] Walter Jones and Tom Medrek. Parallelism ups performance. *EE Times*, October 1999.

[3] Charlie Jenkins. Distributed processing is on top. *EE Times*, January 2000.

[4] Linley Gwennap. Net processor makers race toward 10-gbit/s goal. *EE Times*, June 2000.

[5] Nick McKeown. Network Processors and their Memory. Keynote talk at Workshop on Network Processors and Applications - NP3, February 2004.

[6] Desi Rhoden. The Evolution of DDR. VIA Technology Forum, 2005.

[7] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[8] Charitakis, Anagnostakis, and Markatos. An active traffic splitter architecture for intrusion detection. In *Proceedings of 11th IEEE/ACM MASCOTS*, Orlando, Florida, oct 2003.

[9] Willem de Bruijn, Asia Slowinska, Kees van Reeuwijk, Tomas Hruby, Li Xu, and Herbert Bos. SafeCard: a Gigabit IPS on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, Hamburg, Germany, September 2006.

[10] SNORT group. Snort: The Open Source Network Intrusion Detection System, 2002. http://www.snort.org.

[11] LIBPCAP group. Tcpdump/libpcap, 2002. http://www.tcpdump.org.

[12] NTOP group. Ntop: web-based passive network monitoring application, 2002. http://www.ntop.org.

[13] Ken Keys, David Moore, Ryan Koga, Edouard Lagache, Michael Tesch, and K. Claffy. The architecture of CoralReef: an Internet traffic monitoring software suite. In *PAM2001 — A workshop on Passive and Active Measurements*. CAIDA, RIPE NCC, April 2001. http://www.caida.org/tools/measurement/coralreef/.

[14] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, San Francisco, CA, December 2004.

[15] Stephen Northcutt and Judy Novak. *Network Intrusion Detection: An Analysts' Handbook*. Sams, 3rd edition, September 2002.

[16] R. Puri, Kang-Won Lee, K. Ramchandran, and V. Bharghavan. An integrated source transcoding and congestion control paradigmfor video streaming in the internet. In *IEEE Transactions On Multimedia*, volume 3, pages 18–32, March 2001.

[17] Mihai-Lucian Cristea, Leon Gommans, Li Xu, and Herbert Bos. The Token Based Switch: per-packet access authorisation to optical shortcuts. In *Proceedings of IFIP Networking'07*, Atlanta, GA, USA, May 2007.

[18] Steven McCanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX conference*, San Diego, Ca., January 1993.

[19] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large m essages. In *USENIX Winter*, pages 153–165, 1994.

[20] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.

[21] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM'96*, pages 53–59, 1996.

[22] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.

[23] G. Robert Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Computer Communication Review, ACM SIGCOMM*, October 1998.

[24] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt. Architecture of a network monitor. In *Proceedings of PAM'03*, 2003.

[25] Michael Bellion and Thomas Heinz. HiPAC - high performance packet classification. Technical report, 2005.

[26] Kees van Reeuwijk and Herbert Bos. Ruler: high-speed traffic classification and rewriting using regular expressions. Technical report, Vrije Universiteit Amsterdam, July 2006.

[27] Thomas Hruby, Kees van Reeuwijk, and Herbert Bos. Ruler: easy packet matching and rewriting on network processors. In *Symposium on Architectures for Networking and Communications Systems (ANCS'07)*, Orlando, FL, USA, December 2007.

[28] V. Jacobson, S. McCanne, and C. Leres. *pcap(3) - Packet Capture library*. Lawrence Berkely Laboratory, Berkeley, CA, October 1997.

[29] Tom M. Thomas. *Juniper Networks Reference Guide: JUNOS Routing, Configuration, and Architecture*, chapter Juniper Networks Router Architecture. January 2003.

[30] Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, and Per Gunningberg. Silk: Scout paths in the linux kernel, tr 2002-009. Technical report, Department of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.

[31] Kurt Keutzer Niraj Shah, William Plishker. NP-Click: A programming model for the Intel IXP1200. In *2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, CA*, February 2003.

[32] Andrew T. Campbell, Stephen T. Chou, Michael E. Kounavis, Vassilis D. Stachtos, and John Vicente. NetBind: a binding tool for constructing data paths in network processor-based routers. In *Proceedings of IEEE OPENARCH 2002*, June 2002.

[33] Dr. Jeppe Jessen and Amit Dhir. Programmable Network Processor Platform. Technical report, Xilinx, July 2002.

[34] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, Hamilton, New Zealand, April 2000.

[35] Nicholas Weaver, Vern Paxson, and Jose M. Gonzalez. The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention. In *ACM/SIGDA International Symposium on FPGAs*, Monterey, California, USA, February 2007.

[36] David Anto, Jan Koenek, Kateina Minakov, and Vojtch ehk. Packet header matching in combo6 ipv6 router. Technical Report 1, CESNET, 2003.

[37] Chris Clark, Wenke Lee, David Schimmel, Didier Contis, Mohamed Kone, and Ashley Thomas. A hardware platform for network intrusion detection and prevention. In *The 3rd Workshop on Network Processors and Applications (NP3)*, Madrid, Spain, Feb 2004.

[38] Ioannis Charitakis, Dionisios Pnevmatikatos, and Evangelos Markatos. Code generation for packet header intrusion analysis on the ixp1200 network processor. In *SCOPES 7th International Workshop*, 2003.

[39] Jeffrey C. Mogul. Tcp offload is a dumb idea whose time has come. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[40] J. Apisdorf, k claffy, K. Thompson, and R. Wilder. Oc3mon: Flexible, affordable, high performance statistics collection. In *1996 USENIX LISA X Conference*, pages 97–112,, Chicago, IL, September 1996.

[41] Gianluca Iannaccone, Christophe Diot, Ian Graham, and Nick McKeown. Monitoring very high speed links. In *ACM SIGCOMM Internet Measurement Workshop 2001*, September 2001.

[42] Sriram R. Chelluri, Bryce Mackin, and David Gamba. Fpga-based solutions for storage-area networks. Technical report, Xilinx Inc., March 2006.

[43] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th Symposium on Operating System Principles*, pages 39–51, Austin, Tx., November 1987. ACM.

[44] J. van der Merwe, R. Caceres, Y. Chu, and C. Sreenan. Mmdump - a tool for monitoring internet multimedia traffic. *ACM Computer Communication Review*, 30(4), October 2000.

[45] Sotiris Ioannidis, Kostas G. Anagnostakis, John Ioannidis, and Angelos D. Keromytis. xPF: packet filtering for low-cost network monitoring. In *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR)*, pages 121–126, May 2002.

[46] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Adminstration Conference*, 1999.

[47] Intel Corporation. Intel IXP1200 Network Processor, 2000. http://developer.intel.com/ixa.

[48] Intel Corporation. Intel IXP2xxx Network Processor, 2005. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[49] IETF working group. Internet protocol flow information export. http://www.ietf.org/html.charters/ipfix-charter.html.

[50] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[51] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 40–52. ACM Press, 1996.

[52] Herbert Bos and Bart Samwel. The OKE Corral: Code organisation and reconfiguration at runtime using active linking. In *Proceedings of IWAN'2002*, Zuerich, Sw., December 2002.

[53] Kostas G. Anagnostakis, S. Ioannidis, S. Miltchev, and Michael B. Greenwald. Open packet monitoring on flame: Safety, performance and applications. In *Proceedings of IWAN'02*, Zuerich, Switzerland, December 2002.

[54] Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *Proceedings of OPENARCH'02*, New York, USA, June 2002.

[55] P. Paulin, F. Karim, and P. Bromley. Network processors: a perspective on market requirements, processor architectures and embedded s/w tools. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 420–429, Piscataway, NJ, USA, 2001. IEEE Press.

[56] IBM Microelectronics. The network processor enabling technology for high-performance networking, 1999. http://www-3.ibm.com/chips/products/wired.

[57] Bay Microsystems. Network processors: Chesapeake, montego, biscayne, 2007. http://www.baymicrosystems.com/products/network-processors.html.

[58] LSI Corporation. Former agere systems network processors, 2007. http://www.lsi.com/networking_home/networking_products/network_processors.

[59] Broadcom Corporation. Communications processors, 2007. http://www.broadcom.com/products/Enterprise-Networking/Communications-Processors.

[60] Xelerated. Xelerator x11 network processor, 2007. http://www.xelerated.com.

[61] Ezchip Technologies. Np-2 network processor, 2007. http://www.ezchip.com/html/in_prod.html.

[62] Vitesse Semiconductor. Iq2000 and iq2200 families of network processors, 2002. http://www.vitesse.com/products.

[63] PLCOpen Standardization in Industrial Control Programming. Iec-61131, 2003. http://www.plcopen.org/.

[64] Jonathan T. Moore, Jessica Kornblum Moore, and Scott Nettles. *Active Networks*, volume 2546, chapter Predictable, Lightweight Management Agents, pages 111–119. Springer Berlin / Heidelberg, 2002.

[65] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A compile time based approach for solving out-of-order communication in Kahn Process Networks. In *Proceedings of IEEE 13th International Conference on Application-specific Systems, Architectures and Processors*, July 17-19 2002.

[66] Rhys Weatherley. Treecc: An aspect-oriented approach to writing compilers. *Free Software Magazine*, 2002.

[67] Pty Ltd Southern Storm Software and Inc. Free Software Foundation. Tree compiler-compiler, 2003. http://www.southern-storm.com.au/treecc.html.

[68] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, chapter The Role of Trust Management in Distributed Systems Security, pages 185–210. Springer-Verlag Lecture Notes in Computer Science, Berlin, 1999.

[69] Jeffrey B. Rothman and John Buckman. Which OS is fastest for high-performance network applications? *SysAdmin*, July 2001.

[70] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[71] Tammo Spalink, Scott Karlin, and Larry Peterson. Evaluating network processors in IP forwarding. Technical Report TR-626-00, Department of Computer Science, Princeton University, November 2000.

[72] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. The sidecore approach to efficient virtualization in multicore systems. In *HotOS 2007*, San Diego, USA, May 2007.

[73] Trung Nguyen, Willem de Bruijn, Mihai Cristea, and Herbert Bos. Scalable network monitors for high-speed links: a bottom-up approach. In *Proceedings of IPOM'04*, Beijing, China, 2004.

[74] John W. Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim. An extensible, system-on-programmable-chip, content-aware Internet firewall. In *Field Programmable Logic and Applications (FPL)*, page 14B, Lisbon, Portugal, September 2003.

[75] Bart Kienhuis, Edwin Rypkema, and Ed Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.

[76] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proceedings of 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, September 2003.

[77] Mihai Lucian Cristea, Willem de Bruijn, and Herbert Bos. FPL-3: towards language support for distributed packet processing. In *Proceedings of IFIP Networking 2005*, Waterloo, Canada, May 2005.

[78] Intel Coorp. Internet exchange architecture: Programmable network processors for today's modular networks. White Paper, 2002.

[79] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *In Proceedings of the International World Wide Web Conference, IEEE*, pages 252–262, May 2002.

[80] Panos Trimintzios, Michalis Polychronakis, Antonis Papadogiannakis, Michalis Foukarakis, Evangelos Markatos, and Arne Oslebo. DiMAPI: An Application Programming Interface for Distributed Network Monitoring. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Vancouver, Canada, Apr 2006.

[81] Dror G. Feitelson. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–77, 1990.

[82] T. Simsek and P. Varaiya. Communication and control of distributed hybrid systems. In *Proceedings of 41st IEEE Conference on Decision and Control*, Las Vegas, USA, Dec 2002.

[83] Wei Xu, Joseph L. Hellerstein, Bill Kramer, and David Patterson. Control considerations for scalable event processing. In *DSOM*, pages 233–244, 2005.

[84] X. Li, L. Sha, and X. Zhu. Adaptive control of multi-tiered web applications using queueing predictor. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, pages 106–114. IEEE Press, 2006.

[85] Norman Bobroff and Lily Mummert. Design and implementation of a resource manager in a distributed database system. *Journal of Network and Systems Management*, 13(2):151–174, June 2005.

[86] Cong Du, Xian-He Sun, and Ming Wu. Dynamic scheduling with process migration. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 92–99, Washington, DC, USA, 2007. IEEE Computer Society.

[87] Asser N. Tantawi and Don Towsley. Optimal static load balancing in distributed computer systems. *J. ACM*, 32(2):445–465, 1985.

[88] G.E. Box, G.M. Jenkins, and G.C. Reinsel. *Forecasting and Control*, chapter Time Series Analysis. Prentice-Hall, 3rd edition, 1994.

[89] R. Gibbens and F. Kelly. Measurement-based connection admission control. In *InternationalTeletraffic Congress*, Jun 1997.

[90] Sugih Jamin, Scott Shenker, and Peter B. Danzig. Comparison of measurement-based call admission control algorithms for controlled-load service. In *INFOCOM*, pages 973–980, 1997.

[91] Jingyu Qiu and Edward W. Knightly. Measurement-based admission control with aggregate traffic envelopes. *IEEE/ACM Transactions on Networking*, 9(2):199–210, 2001.

[92] Viktoria Elek, Gunnar Karlsson, and Robert Ronngren. Admission control based on end-to-end measurements. In *INFOCOM*, pages 623–630, 2000.

[93] Jens Milbrandt, Michael Menth, and Jan Junker. Performance of experience-based admission control in the presence of traffic changes. In *Fifth IFIP Networking Conference (NETWORKING 2006)*, Coimbra, Portugal, 5 2006.

[94] M. Ghaderi, J. Capka, and R. Boutaba. Prediction-based admission control for diffserv. *Vehicular Technology Conference*, 3:1974–1978, Oct 2003.

[95] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, pages 183–193, New York, NY, USA, 1993. ACM Press.

[96] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Trans. Networking*, 5(1):71–86, 1997.

[97] Yi Qiao, Jason Skicewicz, and Peter Dinda. An empirical study of the multiscale predictability of network traffic. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 66–76, Washington, DC, USA, 2004. IEEE Computer Society.

[98] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[99] Jamie Callan. *Advances in information retrieval*, chapter 5: Distributed Information Retrieval, pages 127–150. Kluwer Academic Publishers, 2000.

[100] Yuliy Baryshnikov, Ed G. Coffman, Guillaume Pierre, Dan Rubenstein, Mark Squillante, and Teddy Yimwadsana. Predictability of web-server traffic congestion. In *Proceedings of the Tenth IEEE International Workshop on Web Content Caching and Distribution*, pages 97–103, September 2005. http://www.globule.org/publi/PWSTC_wcw2005.html.

[101] SURFnet builds hybrid optical and packet switching network. Lightwave Online Article, March 2004. http://lw.pennnet.com/Articles/Article_Display.cfm?Section=OnlineArticles &SubSection=Display&PUBLICATION_ID=13&ARTICLE_ID=201399.

[102] Linda Winkler and HOPI_Design_Team. The hybrid optical and packet infrastructure (HOPI) testbed. Internet2 whitepaper, April 2004. http://networks.internet2.edu/hopi/hopi-documentation.html.

[103] J. Vollbrecht, P. Calhoun, S. Farrel, Leon Gommans, G. Gross, B. Bruin, Cees de Laat, M. Holdrege, and D. Spence. RFC2904, AAA Authorization Framework. IETF, 2000.

[104] Leon Gommans, Franco Travostino, John Vollbrecht, Cees de Laat, and Robert Meijer. Token-based authorization of connection oriented network resources. In *Proceedings of GRIDNETS*, San Jose, CA, USA, Oct 2004.

[105] M. Bellare, R. Canetti, and H. Krawczyk. RFC2104, HMAC: Keyed-Hashing for Message Authentication. IETF, 1997.

[106] R. Boutaba, Y. Iraqi, and A. Ghlamallah. User controlled lightpaths architecture design. Presentation, May 2003.

[107] S. Figueira, S. Naiksatam, and H. Cohen. DWDM-RAM: Enabling Grid Services with Dynamic Optical Networks. In *Proceedings of the SuperComputing Conference (SC2003)*, Phoenix, Arizona, Nov 2003.

[108] Haining Wang, Abhijit Bose, Mohamed El-Gendy, and Kang G. Shin. IP Easy-pass: a light-weight network-edge resource access control. *IEEE/ACM Transactions on Networking*, 13(6):1247–1260, 2005.

[109] S. Blake, D.Black, M. Carlson, E. Davies, Z. Wang, and W. Weis. RFC2475, An Architecture for Differentiated Services. IETF, 1998.

[110] E. Rosen, A. Viswanathan, and R. Callon. RFC3031, Multiprotocol Label Switching Architecture. IETF, 2001.

[111] Raouf Boutaba, Wojciech M. Golab, Youssef Iraqi, Tianshu Li, and Bill St. Arnaud. Grid-Controlled Lightpaths for High Performance Grid Applications. *J. Grid Comput.*, 1(4):387–394, 2003.

[112] Tom DeFanti, Cees de Laat, Joe Mambretti, Kees Neggers, and Bill St. Arnaud. TransLight: a global-scale LambdaGrid for e-science. *Commun. ACM*, 46(11):34–41, 2003.

[113] C. de Laat, E. Radius, and S. Wallace. The rationale of current optical networking initiatives. *Future Generation Computer Systems*, 19(6):999–1008, Aug 2003.

[114] L. Gommans, F. Dijkstra, C. de Laat, A. Taal, A. Wan, Lavian T., I. Monga, and F. Travostino. Applications drive secure lightpath creation across heterogeneous domains. *IEEE Communications Magazie*, 44(3):100–106, March 2006.

[115] Ajay Tirumala et al. *Iperf tutorial*. http://iperf.sourceforge.net.

[116] J. Wu, S. Campbell, J.M. Savoie, H. Zhang, G.v. Bochmann, and B.St. Arnaud. User-managed end-to-end lightpath provisioning over ca*net 4. In *Proceedings of the National Fiber Optic Engineers Conference*, Orlando, FL, USA, Sep 2003.

[117] D.L. Truong, O. Cherkaoui, H. ElBiaze, N. Rico, and M. Aboulhamid. A Policy-based approach for User Controlled Lightpath Provisioning. In *Proceedings of NOMS*, Seoul, Korea, Apr 2004.

[118] S. Figueira, S. Naiksatam, and H. Cohen. OMNInet: a Metropolitan 10Gb/s DWDM Photonic Switched Network Trial. In *Proceedings of Optical Fiber Communication*, Los Angeles, USA, Feb 2004.

[119] Tom Lehman, Jerry Sobieski, and Bijan Jabbari. DRAGON: A Framework for Service Provisioning in Heterogeneous Grid Networks. *IEEE Communications Magazine*, 44(3), March 2006.

Samenvatting

In een vereenvoudigde weergave is het Internet te zien als een wereldwijd netwerk dat bestaat uit knopen (plaatsen van aanvoer waar signalen naar verschillende richtingen worden uitgezonden) die met elkaar verbonden zijn via verbindingen en schakels. Applicaties – zoals websites of netwerkspellen – die op deze knopen draaien, zijn op een virtuele manier met elkaar verbonden met behulp van dataverkeer dat door deze fysieke verbindingen en hun schakels stroomt. Nieuwe computers, nieuwe applicaties zoals videofoons of IPTV en hun gebruikers, voegen zich dagelijks bij het Internet. Het Internet is een onoverzichtelijke plek geworden waar goede en slechte dingen gebeuren, omdat er geen controle is over het gehele Internet. Het is nodig dat iedere gebruiker of corporatie zelf zijn of haar knoop die verbonden is met het Internet, beveiligt. Om de verspreiding van slechte dingen naar bepaalde computers of een groep van computers (bijvoorbeeld kleine netwerken zoals thuis- of bedrijfsnetwerken) te voorkomen, is een soort 'stethoscoop' nodig die het netwerkverkeer dat onze privé-domeinen binnenkomt, inspecteert. Een dergelijke stethoscoop voor netwerkverkeer is een *traffic monitoring system*.

Een *traffic monitoring system* inspecteert – naast vele andere taken – het netwerkverkeer op verboden inhoud zoals viruspatronen of expliciete seksuele inhoud. Wanneer een verboden inhoud ontdekt is, zal het systeem de gebruiker waarschuwen, of het 'slechte' verkeer simpelweg laten verdwijnen. Een *traffic monitoring system* kan zich bevinden in de schakels waar de *personal computers* (pc's) of de kleine netwerken van computers met het Internet verbonden zijn.

Voor de beveiliging van pc's inspecteert een *traffic monitoring system* het dataverkeer met een snelheid van tientallen Kbps tot honderd Mbps. Bij het beschermen van een bedrijfsnetwerk krijgt een *traffic monitoring system* te maken met hogesnelheidsverkeer van multi-gigabits/sec. Voor netwerkverbindingen met lage snelheden volstaan goedkope software oplossingen om het verkeer te inspecteren, zoals firewalls op pc's. Hoge snelheidsverbindingen vergen echter gespecialiseerde hardware en software.

De huidige hardwareoplossingen voor het verwerken van dataverkeer (zoals netwerkprocessors) bieden controleapplicaties bij extreem hoge snelheden slechts genoeg verwerkingscapaciteit om een deel van het verkeer te inspecteren. Hierdoor kunnen bijvoorbeeld alleen de headers van de datapakketten, waaruit netwerkverkeer is opgebouwd, verwerkt worden. Vanwege de continue vernieuwingen op het gebied van malafide activiteiten of virussen, bijvoorbeeld het

gebruik maken van verschillende delen van de pakketten, zijn echter verwerkingssystemen nodig die in staat zijn om het gehele pakket met hoge snelheid te verwerken. Een verwerkingssysteem voor dataverkeer dat in staat is om het gehele pakket met multi-gigabitsnelheid te verwerken, vereist specifieke architecturen om de technologische kloof tussen de verwerkingssnelheid van processors (elektronen) en de netwerksnelheid van glasvezel (licht) te dichten. Hoewel de huidige architecturen al gebruik maken van meerdere processors om dataverkeer parallel te verwerken, ben ik van mening dat enkel een parallelle benadering niet genoeg is om een duurzame oplossing te bieden voor de huidige en toekomstige eisen.

Dit onderzoek presenteert een parallelle en gedistribueerde aanpak om netwerkverkeer met hoge snelheden te verwerken. De voorgestelde architectuur biedt de verwerkingscapaciteit die vereist is om één of meer dataverkeerverwerkende toepassingen (bijvoorbeeld *traffic monitoring*) met hoge snelheden te laten draaien door het verwerken van gehele pakketten met multi-gigabitsnelheden onder gebruikmaking van een parallelle en gedistribueerde verwerkingsomgeving. Bovendien is de architectuur flexibel en zal aan toekomstige eisen kunnen voldoen door heterogene verwerkingsknopen te ondersteunen, zoals verschillende hardwarearchitecturen of verschillende generaties van eenzelfde hardwarearchitectuur. Naast de verwerkings-, flexibele en duurzame eigenschappen, verschaft de architectuur een gebruiksvriendelijke omgeving door de hulp van een nieuwe programmeertaal, FPL genaamd, voor verkeersverwerking in een gedistribueerde omgeving.

Om een flexibel en duurzaam verwerkingssysteem te creëren, maakt de architectuur gebruik van twee principes: parallellisme en distributie. Ten eerste wordt parallellisme gebruikt binnen een enkele verwerkingsknoop om op één van de volgende manieren de binnenkomende pakketten parallel te verwerken: meervoudige toepassingen of taken kunnen hetzelfde pakket parallel verwerken, één toepassing kan meervoudige pakketten parallel verwerken, of een combinatie van deze twee mogelijkheden. De implementatie van de voorgestelde architectuur maakt gebruik van parallellisme in elk van de drie meest gebruikte en ondersteunde hardwarearchitecturen: pc's, netwerkprocessors en FPGA's (*Field Programmable Gate Arrays*, herconfigureerbare hardware). In pc's profiteer ik bijvoorbeeld van *time-shared* parallellisme dat aangeboden wordt door het besturingssysteem. In netwerkprocessors, speciaal ontworpen hardware met multi-cores en multi-memories, maak ik gebruik van parallellisme dat door de hardware ondersteund wordt. In FPGA systemen vervaardig ik een op maat gemaakt parallel 'apparaat', door de intensieve verwerkingstaken te draaien op individueel aangepaste hardwarekernen, terwijl één kern het proces beheert. Ten tweede distribueer ik de verwerkingsapplicaties van dataverkeer die meer verwerkingscapaciteit eisen dan een enkele knoop kan bieden.

Wanneer ik een verwerkingsapplicatie van dataverkeer distribueer, krijg ik met de volgende twee aspecten te maken: de verdeling van taken en de verdeling van dataverkeer dat verwerkt moet worden. Aan de ene kant wordt het splitsen van een dataverkeerapplicatie in taken – die dan verdeeld worden over de gedistribueerde architectuur – gedaan met behulp van specifieke constructies in de voorgestelde programmeertaal om datapakketten te verwerken. De taken worden dan vervolgens vertaald naar een code voor ieder van de beoogde hardwareplatformen. Aan de andere kant wordt in de verdeling van dataverkeer voorzien door een nieuw concept waarbij een verwerkingsknoop in de gedistribueerde architectuur taken ondersteunt met dataverkeerverwerking als doel, taken met dataverkeerverdeling of routing als doel, of taken met zowel dataverkeerverwerking als -verdeling tegelijk als doel.

Dit concept van een volledig programmeerbare verwerkingsknoop wordt geïmplementeerd met behulp van dezelfde FPL taal om datapakketten te verwerken. Deze taal biedt toegang tot het laagste dataniveau: ruwe data van het netwerkverkeer. In de gedistribueerde architectuur stel ik een hiërarchische topologie voor, waar de eerste knopen in de verwerkingshiërarchie ook dataverkeerverdeling verrichten naar de volgende knopen en waar de knopen in de laatste regionen alleen specifieke verkeersverwerking uitvoeren. Met andere woorden, hoe verder er afgedaald wordt in de verwerkingshiërarchie, hoe minder dataverkeer er herverdeeld hoeft te worden en daardoor voeren de laatstgenoemde verwerkingsknopen intensievere en meer gespecialiseerde taken uit om het dataverkeer te verwerken.

Ten slotte heb ik ontdekt dat een dergelijk complex gedistribueerd systeem werkt in een dynamische omgeving (waarin het binnenkomende verkeer bijvoorbeeld varieert in de loop van de tijd) en dat het een controlemechanisme nodig heeft om het systeem stabiel te houden en onbemand te laten draaien. Daarom ontwierp ik een extensie voor de gedistribueerde architectuur met beheer als doel. Hoewel er diverse manieren van beheer bestaan (bijvoorbeeld gecentraliseerd of gedistribueerd), verwezenlijkte ik alleen een gecentraliseerde aanpak vanwege de kleine schaal (tientallen knopen) van mijn gedistribueerde architectuur voor *traffic monitoring*.

Ik ben van mening dat een oplossing voor het probleem van verwerking van dataverkeer met hoge snelheden moet bestaan uit een parallelle en gedistribueerde aanpak, waarbij ook gezorgd moet worden voor heterogeniteit, controle en gebruiksgemak.

Acknowledgments

First of all, I am grateful to my parents for the chances to grow according to my talents and preferences. They taught me the right ideals for life and supported me in all my trials. I also thank my wife, Violeta, for her understanding. She often had to share me with Science during leisure times and to miss my presence during many conference stays.

A big contribution to the friendly atmosphere at LIACS was due to the discussions and work with my colleagues: Claudiu Zissulescu, Alex Turjan, Dmitry Cheresiz, Laurentiu Nicolae, Ioan Cimpian, Willem de Bruijn, Todor Stefanov. In addition, I mention Lennert Buytenhek for his invaluable help in Linux world. I also thank to Trung Nguyen and Li Xu for the collaboration we had during their master projects.

I will always remember the moment I have chosen an academic life being inspired by the scientific view on systems of Prof. dr. Emil Ceanga. I also thank him for showing me the first steps in research during my master project.

I am obliged to Associate Prof. dr. Nicu Sebe because he re-oriented my engineering career to research by introducing me to Herbert's group.

I am grateful to my friend, Mirona Elena Ciocîrlie, for her reviews on this thesis and advices during our elementary and secondary school period.

My final thank-you is directed to my grandfather, Tătăica, a well-educated peasant who fought in WWII and had enough resources left to train my thoughts through long discussions during the summer holidays of my childhood. Looking back in time, I remember that Tătăica has foreseen that I would be a doctor in science, 25 years before I found the meaning.

Curriculum Vitae

Mihai Cristea was born in Galaţi, România in 1976. After graduating from Mihail Kogălniceanu High School in 1994, he started his studies at the Faculty of Shipbuilding and Electrical Engineering. He received his B.Sc. degree in Automation and Computer Science Engineering in 1999 and the M.Sc. degree in Artificial Intelligence in Process Control in 2000 from the "Dunărea de Jos" University of Galaţi.

After half a year of working in automation engineering at ArcelorMittal steel making factory in Galaţi he went for software development applied to industrial fields such as marine engineering to two companies (Romanian and Dutch) for about two years.

In the fall of 2002, Mihai Cristea chose an academic career and joined as a Ph.D. student the High Performance Computing group of the Computer Science Department of Leiden University headed by Prof. Dr. H.A.G Wijshoff. The Ph.D. research was part of a joint project between this group and the group of Dr. Herbert Bos at the Vrije Universiteit, Amsterdam.