VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DESIGN OF PROBE FOR FLOW BASED MONITOR-ING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE LUKÁŠ SOĽANKA
AUTHOR

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH ARCHITEKTURY SONDY PRO MONITOROVÁNÍ SÍŤOVÝCH TOKŮ

DESIGN OF PROBE FOR FLOW BASED MONITORING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE LUKÁŠ SOĽANKA
AUTHOR

VEDOUCÍ PRÁCE Ing. JAN KOŘENEK
SUPERVISOR

BRNO 2009

Abstrakt

Cílem této práce je návrh a implementace síťové monitorovací sondy založené na konceptu toků. Monitorování je rozděleno na hardwarovou část, která je schopná zpracovávat data na vysokých rychlostech, a na softwarovou část, která zabezpečuje vysokou kapacitu paměti potřebnou pro uchování toků. Práce zahrnuje také analýzu a simulace, které ukazují, že tento koncept poskytuje mnoho výhod oproti čistě softwarovým řešením. Navržená sonda pracuje s použitím hardwarového akcelerátoru, poskytuje vysoký výkon a umožňuje uživateli definovat svoji vlastní strukturu záznamu pro monitorování, čímž zabezpečuje vysokou flexibilitu. Systém byl implementován a důkladně otestován monitorováním univerzitní sítě. Je proto připraven pro dlouhodobé použití za účelem monitorování provozu, klasifikace protokolů, detekce anomálií a útoků a mnoha jiných aspektů sítí.

Abstract

This thesis deals with design and implementation of a flow based monitoring probe. The monitoring task performed by the probe is divided into hardware layer, which is capable of measurement at high packet rates, and software layer, which provides large memory for flow storage. Analysis done in the work shows that this concept offers many advantages when compared to software based flow monitoring applications. The probe is designed to be used with a hardware accelerator card and offers high flexibility and performance by a way of user defined monitoring process. The designed system has been implemented and thoroughly tested and is ready for deployment for tasks such as operational monitoring, network traffic classification, anomalies and attacks detection and many others.

Klíčová slova

Síť, IP tok, monitorování, NetFlow, IPFIX

Keywords

Network, IP flow, monitoring, NetFlow, IPFIX

Citace

Lukáš Soľanka: Design of Probe for Flow Based Monitoring, diplomová práce, Brno, FIT VUT v Brně, 2009

Design of Probe for Flow Based Monitoring

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Kořenka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Lukáš Soľanka
26.5.2009

Poděkování

Rád bych poděkoval panu Ing. Janovi Kořenkovi za pomoc při psaní této práce. Jeho optimizmus byl i mojí motivací.

Contents

Chapter 1

Introduction

Internet has currently become a widespread means of communication. Number of users and throughput of communication links are quickly growing as the system becomes ubiquitous. With the ongoing pace of networking growth and spread, a network monitoring infrastructure has become a necessity, requiring devices which collect information about the state of a network. Moreover, because it is a dynamic system, variety of means are needed to capture information about communication between end nodes or other active parts.

Network monitoring has thus been essential way to keep a network functional and provide administrators with knowledge about its state.

Monitoring can be simply defined as a means of collecting data about the state of a network for purpose of analyzing its behavior, limiting unwanted communication, detecting traffic anomalies, malicious behavior or attacks. There are several ways to accomplish this task. One of them is to insert data into traffic deliberately and monitor the behavior of network nodes as the data passes through the network. Such type is called active monitoring. Active measurement can provide useful insight into the structure of network and can help trace a source of problem in case part of the network fails. However its usage is fairly limited, because it does not provide insight into data structures and communication between entities inside the network.

Passive monitoring, on the other hand, gathers information about information flow through a network, by means of capturing data and analyzing it. Again, there are several ways of how to collect this information. One of the simplest is only storage of packet traces into some media. While it does not require any significant amount of work from a user, with high rate networks, system throughput limits the usage of such system. Moreover, long lasting traces require significant amount of storage space. The issue may be solved by means of compression. Only a relevant information about the traffic is stored and further analyzed. Beside classical compression, which does not provide feasible compression rates, another approach might be used. It is based on a notion of communication between two entities on a network. As is widely known, network devices are addressed at several levels, ranging from link layer giving address to endpoint devices within local network, to transport layer, where the processes are distinguished. Addressing is the fundamental concept of the Internet and can be used to distinguish communication *flows* between network devices. The notion of flow has been developed as a Netflow protocol [6], in which the flow is defined as a set of packets passing a measurement point, possessing some measurable common properties. These properties are generally IP source and destination addresses, UDP or TCP source and destination ports and a transport layer protocol type. Thus, this basic definition is behind the collection of data and their classification with respect to communicating entities. Among the common properties, each flow also carries statistical or other user defined information about the data flowing between two nodes of a communication channel.

The way of passive network measurement and the notion of flow is the fundamental concept of this thesis. Flow can be further generalized with respect to the emerging IPFIX standard [8] based on Netflow protocol version 9. It defines a flow not statically as Netflow version 5, but fundamentally as a set of packets passing and observation point of a monitoring device, which share some common properties. The standard defines all these possible properties and their extensions and the infrastructure of an IPFIX compliant monitor.

The main aim of this thesis is to design and implement an IPFIX compliant monitoring device which can handle high-rate networks and provide as much information about the traffic flowing through them and passing a measurement point where the device is connected. The target network rates are 10 Gb/s and more, depending on underlying hardware limits and character of their network interfaces. One of the goals is to provide flow statistical data without packet loss, which, in many cases is crucial for upper layers of a flow based monitoring system. While the system must be tuned up for high rate networks, it is essential for it to be user-configurable as much as possible. The reasons for it are diverse: not only IPFIX defines a whole range of possible flow indicators[1] and it is extensible, but administrators may want to define their own monitoring process, which is completely different from what IPFIX offers. Thus, a great emphasis should be put on flexibility of the design and implementation of such monitoring appliance.

There are several ways on how to design and implement a flow monitoring probe and several working devices already exist that can collect flow data and even maintain and provide useful analysis of these flow traces. One of the fundamental and starting point is the Netflow capability in Cisco routers. Since a router is a switching point in the network, it might be feasible to provide flow collection and their export to other devices. Cisco IOS operating systems therefore offer the capability of Netflow export. However, as the rates of network links approach 10 Gb/s, these routers are no longer able to handle incoming traffic and therefore sampling must be used. Usage of sampling is basically tolerable, but there are situations where it is not feasible, namely in attacks or anomalies detection. Also such simple application as usage based pricing also tries to avoid sampling, because when used, difficult and possibly incorrect traffic reconstruction must be performed on the statistical data, which might not be accurate and thus fair to the provider's customers.

Also, purely software and hardware based flow monitoring devices exist, both types having their strengths and weaknesses. Software offers a short design and implementation cycle, can be easily maintained and reprogrammed and thus is appropriate for prototyping and basic usage. However when dealing with rate beyond 10 Gb/s, purely software based probes might not be able to process the traffic, especially with short packets or in case of attacks or network anomalies. For this purpose, hardware acceleration might be necessary which is able to process data at high packet rates. The upper software layer is then responsible only for flow export and configuration, as it is not a time critical task. Indeed, there exist many flow monitoring probes capable of high speed processing, among Cisco devices, nProbe [11] may be given as an example of a software based probe and the FlowMon probe [46] as an example of a pure hardware based probe.

Although for high-speed networks it might seem appropriate to put whole monitoring process into hardware and use a software part only for configuration, it causes several difficulties. Firstly the design and implementation phase are very long, because mostly the

[1]Packet header field used to distinguish between flows.

available algorithms are suitable rather for software implementations and not for hardware. A good example might be a double linked list used in [46] for flow maintenance which is perfectly suitable in software implementations, but causes many difficulties with design and implementation in hardware. Secondly, such system is very little extensible and also difficult to maintain, possibly incuring several time penalties for developers.

In this thesis, it was opted for a compromise and the two previously mentioned layers, hardware and software, are used jointly in order to keep the hardware part simple, yet still providing enough performance to handle high packet rates. The system is based on a two level aggregation, in which the hardware part preaggregates data at high-speeds, and the software layer handles the remaining data flow to provide as much information compression as possible, all without packet loss if possible.

The structure of the thesis is organized as follows. Chapter 2 provides basic knowledge of the network layered structure. The current fundamental concept is a layered structure with TCP/IP networking model. The chapter describes all four layers of the model and presents protocol data units definitions to the reader. In Chapter 3, network measurement principles are described, dividing the means of monitoring into passive and active measurement. The most important part for this thesis provides an overview of flow based monitoring standards, Netflow and IPFIX. Both chapters are a basic background knowledge necessary to understand the purpose of flow monitoring. Chapter 4 presents a generic monitoring process architecture and describes its basic blocks, all necessary for proper functionality: packet capturing and preprocessing part, flow lookup mechanism, flow update process and flow maintenance.

In order to derive necessary level of complexity of hardware and software layer, Chapter 5 describes an analysis of an existing software probe along with indexing algorithms from Chapter 4. A profiling method is used to assess the limits of a highly optimized pure software solution and maximal packet rate is derived from the profiling results. These results serve as a basis in estimation of requirements on the hardware part. To keep the hardware as simple as possible, the limits on hardware aggregation are estimated in order to assess if it is possible to sufficiently preaggregate data into flows, in order not to overload the higher, software part.

Chapter 6 is a description of the designed architecture on a system level. It should also be noted, that only the hardware part has been considered in this thesis, without the actual software layer (except the design generator), which is the aim of another work [44]. Chapter 7 presents implementation details and a few testing results of the implemented device. Finally, Chapter 8 is the conclusion of the thesis.

Chapter 2

Networking background

When creating a monitoring device, it is essential to be familiar with underlying principles in the network domain. For network monitoring, the process of communication between two end-nodes is of the main interest and measurement devices gather as much information as possible about their interaction. Several protocols in the TCP/IP stack have been designed and implemented and there exist protocols intended for collecting such information as well. Therefore, main aspects of network communication are discussed in next sections, with emphasis on the widely used TCP/IP model and network monitoring protocols.

2.1 TCP/IP networking model

The TCP/IP model is a specification for network protocols used in computer communication. It is derived from the ISO/OSI reference model, but does not provide such a strict layered model. The main architectural principles are *end-to-end principle*, i.e. the most of the network intelligence is concentrated in devices at the end of the network and the core focuses on speed and simplicity. The other one is the *robustness principle*, which states that a system must produce well-formed datagrams, but must accept any datagram that it can interpret [33].

The protocol suite is designed with a layered architecture, forming a communication abstraction. Each layer uses interface from the lower one and provides interface for the upper one. This technique ensures that higher layers do not need to consider details about the underlying architecture (for instance the type of transfer medium used). RFC 1122 [2] defines four layers, illustrated in Figure 2.1, together with the ISO/OSI model. Next sections discuss this topic in more detail, considering also the most important protocols for each layer.

2.1.1 Link Layer

Link layer defines communication procedures over a local network, to which a host is connected. It spans the physical and link layer of the ISO/OSI model and is used to provide abstraction of the hardware to upper layers. Hosts on this layer are generally referred to as *nodes*, that communicate over a specific *link*. Link layer can only span a specific local network, in comparison for example to the network layer, where protocol data unit (PDU) is delivered from source to destination end-node across several connected networks.

A link-layer protocol defines the format of PDU, which can be different depending on the specific protocol used. Here we are mainly interested in the Ethernet II format, the

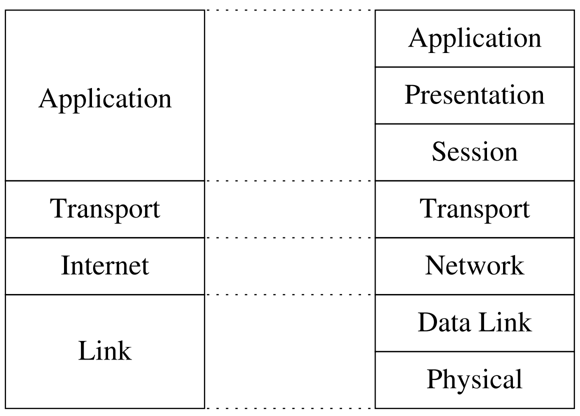


Figure 2.1: TCP/IP and ISO/OSI comparison.

most widely deployed. The frame format is described in the following text and illustrated in Figure 2.2:

- *Destination* and *source MAC* address define the nodes' identification on the link
- 16 bits long *Type/Length* field identifies the upper layer protocol. This can be 0x0800 for an IP datagram, 0x0806 for an ARP packet, 0x8100 for an IEEE 802.1Q VLAN frame or even 0x8847 for MPLS labeled frames (unicast). Note that this field also distinguishes between 802.3 frame and Ethernet II frame: for Ethernet II this field must be greater than 1536.
- The next part is the payload data and it must be between 46 and 1500 bytes. If the data is shorter it must be filled with a padding to meet the range criterion.
- The last section, 4 bytes long CRC checksum provides for error detection in the frame.

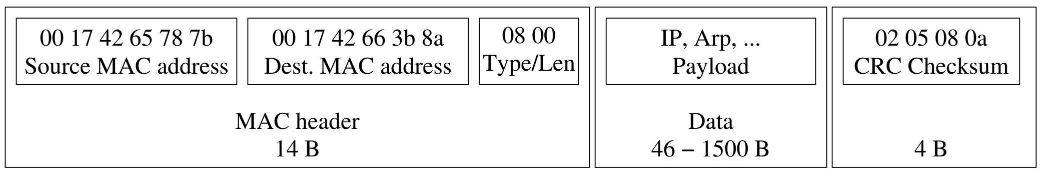


Figure 2.2: Ethernet II frame format

When monitoring 802.1Q networks, the frame structure described above is modified by a VLAN tag. As stated in [40], the purpose of tagging allows a) segregation of frames assigned to different VLANs b) to convey priority with the frame when using IEEE 802 LAN media access control methods that provide no inherent priority capability. It is inserted between the source MAC address and the Type/Length field of the frame. For monitoring network state and communication the VLAN ID field which identifies different collision domains within one segment might be interesting.

When the worst case is considered, i.e. the shortest link frames, with preamble (8 bytes) and inter-frame gap (96 bit times/12 bytes) lengths added, it accounts for 672 bits in total.

Ethernet type	Frame time [ns]	Packet rate [Mpps]
1 Gb/s	672	1.488
10 Gb/s	67.2	14.88
40 Gb/s	16.8	59.52
100 Gb/s	6.72	148.8

Table 2.1: Frame times and packet rates for 1-100 Gb/s Ethernet

The time needed to process one frame (datagram) would therefore be approximately 1400 clock cycles on a 2 GHz processor. It is apparent that on 10 Gbps rates the time for one packet processing is getting seriously low and with emerging 40 and 100 Gb Ethernet standards (Table 2.1) parallelizing the process of packet handling will be necessary.

2.1.2 Internet layer

According to [20] this can be also referred to as *Network layer.* Its purpose is to move packets between end-nodes, possibly across several connected networks. If we restrict the description to the packet-switched schema (vs. circuit switched), the two main operations are:

- *Path determination.* It ensures that correct path is found for the packet to be delivered. At the sender's side the packet is stamped by the receiver's address and must be preserved throughout the path.
- *Packet forwarding,* which determines an address of the next-hop node for the data to be properly delivered to recipient.

Internet layer defines three protocols used [2]: IP [33], ICMP [32] and IGMP [10]. The most important, IP, will be described here, as it defines end-to-end communication mechanisms which are required for the proper monitoring activity. It implements two basic functions: *addressing* and *fragmentation* and is often characterized as the best-effort system, i.e. there is no guarantee that the communication will be reliable or that the user gets specific amount of resources reserved for the delivery of datagrams to maintain quality of service.

IP protocol is currently defined for two versions, IP version 4 (IPv4) and IP version 6 (IPv6), both depicted in Figure 2.3. IPv4 header contains following fields:

- *Version* field discriminates between the versions of the protocol, as stated above.
- *Internet Header Length* (IHL) is the length of the header, in 32 bit words.
- *Type of Service* (TOS) is for traffic differentiation.
- *Total Length* is the lengths of datagram, in octets.
- *Identification*, *Flags* and *Fragment Offset* fields provide for the datagram reassembling
- *Time to Live* (TTL) value defines the maximum number of time the datagram is allowed to remain in the network.
- *Protocol* field defines the next level protocol carried in the data portion of the datagram.

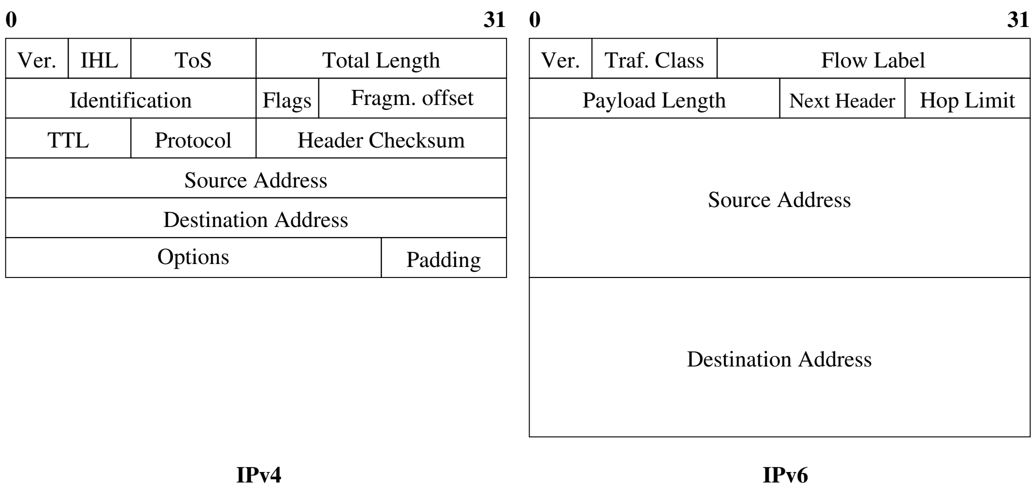


Figure 2.3: IP version 4 and 6 datagram format.

- *Header Checksum*
- *Source Address* and *Destination Address* of end-hosts.

IPv6 emerged in response to the possible exhaustion of network addresses and the fast growing of routing tables. The IP new generation (IPng) has been designed [3] and the protocol simplified in comparison to the former version. The header (without any extension headers) is defined as follows:

- *Version*
- *Flow Label* is used by a host to identify datagrams that are to be treated specially by routers.
- *Payload Length*, in octets
- *Next Header*, is the same as *Protocol* field in IPv4.
- *Hop limit* is equivalent to the TTL field.
- *Source Address* and *Destination Address* are equivalent to IPv4, except the length of the field is 128 bits.

We can see in Figure 2.3 that the header structure is fixed without any options. Embedding some specific information into the IPv6 header can be done by specifying *Extension Headers* (discussed in more detail in [3]).

2.1.3 Transport layer

The purpose of the transport layer is to ensure logical communication between application processes running on different hosts. This is in contrast with the network layer, where the primary aim is best-effort delivery of data between end-hosts.

There are two protocols available for the application layer, distinguishable by the type of the service they provide. The first one, *UDP* is connectionless and provides no guarantee

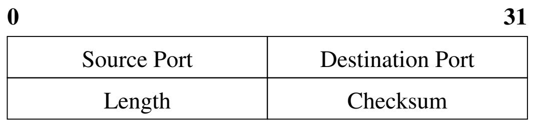


Figure 2.4: UDP segment format

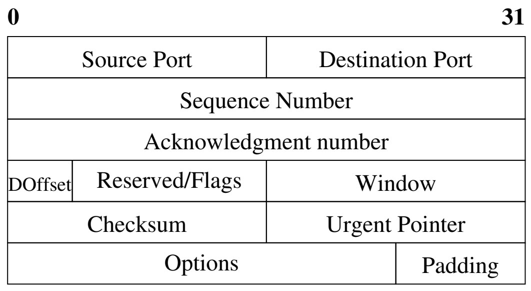


Figure 2.5: TCP segment format

that the data will be delivered. The other one, connection-oriented *TCP* provides reliable data transfer with congestion control. We will keep here a convention from [20] and will refer to the protocol data unit for the transport layer as a *segment*, for both TCP and UDP.

Main aim of UDP [31] is to provide a procedure for an application processes with a minimum protocol overhead. Therefore the segment structure (Figure 2.4) is fairly simple. It contains only 16 bit *Source* and *Destination Port*, 16 bit *Length* and *Checksum* fields, followed by data octets. UDP does not ensure any reliable delivery of messages, nor any in-order delivery or retransmission and thus its main use is for applications, where data loss does not cause their fatal failure.

TCP, on the other hand, provides reliable, end-to-end connectivity between processes [34]. It assumes that the underlying services are potentially unreliable and thus is very robust. The resources to ensure this service are a) *Basic Data Transfer* for duplex transmission between end-processes b) *Reliability* to recover from data damage c) *Flow Control* for the receiver to control amount of data sent by the sender d) *Multiplexing* to allow processes at a single host to share the connection e) *Precedence* and *Security* to be indicated by users of the connection.

Complete description of the TCP segment format is in Figure 2.5. Only the most important fields will be described here:

- *Source* and *Destination Port* numbers identify processes at a single host
- *Sequence Number* is the sequence number of the first data octet present in the segment
- *Acknowledgment Number* is the value of the next sequence number the sender of the segment is expecting to receive. This is in conjunction with the *ACK* flag and provides for in-order data delivery or indicates to the opposite end-process that the data might have been lost.
- *Control Bits* are necessary for the communication mechanisms used by TCP. The bits may be monitored in the form of aggregate statistics.
- *Window* field is the number of data octets the sender of the segment is able to accept.
- *Checksum* field provides for error detection.

2.1.4 Application layer

The top level of the TCP/IP model is a direct interface to applications. Compared to ISO/OSI, it spans the Session, Presentation and Application layers. It defines higher-level protocols for the applications' communication. Examples of such protocols are File Transfer Protocol (FTP) or Hyper Text Transfer Protocol (HTTPS).

Basically, applications communicate via application protocols conforming to the client-server model. A server listens on the specific address, determined by a pair of IP address:Port. When the client needs to communicate with server, it initiates connection based upon underlying layers' protocols. Both TCP and UDP may be used, depending on the requirements for reliability of the communication.

Chapter 3

Network measurement

In the previous chapter we have seen that network traffic contains a lot of important information, which is necessary for functionality and performance of today's infrastructure. Determining the state of the network and of connected devices might be crucial for keeping the whole system in consistent state. Thus, measurement can be defined as a periodic activity of determining the state of the nodes in the network and collecting information about the communication of nodes in it. Several means of gathering and analyzing the information that can be extracted from the network will be discussed in this chapter.

3.1 Active network measurement

Active measurement provides end-to-end performance evaluation. An end-device sends packet probes and while the packets traverse through the network to reach the destination host, the behavior of active devices is saved or analyzed.

One of the uses of active packet probes is in delay estimation and topology scanning. The well-known tools for such estimation include, among any others, `ping` or `traceroute`. Another aspect is the bandwidth estimation, which might be important, for instance, to ensure Quality of Service (QoS). Several tools can be named: iperf tool [43] for TCP/UDP throughput, SProbe [39] for end-to-end bandwidth estimation and many others.

While active measurement provides useful insight into the bandwidth and topology organization, it does not gather information about the actual traffic that flows in the network. Therefore its usage in this thesis is fairly limited and it has been included only for completeness.

3.2 Passive network measurement

Passive measurement does not alter traffic by insertion of any data into the network. A monitoring device rather collects observed packets and stores them or maintains statistics database about them. Collected information may be then further analyzed. We will now discuss three basic categories of passive measurement techniques used.

Packet measurement provides fine-grained information about the state of the network. Packets are simply copied and stored for further analysis or the stream is monitored on-line. Although packet traces provide maximum amount of information they are very demanding in terms of consumed resources for packets storage and require monitoring device to cope with the ever-increasing rates of high speed links.

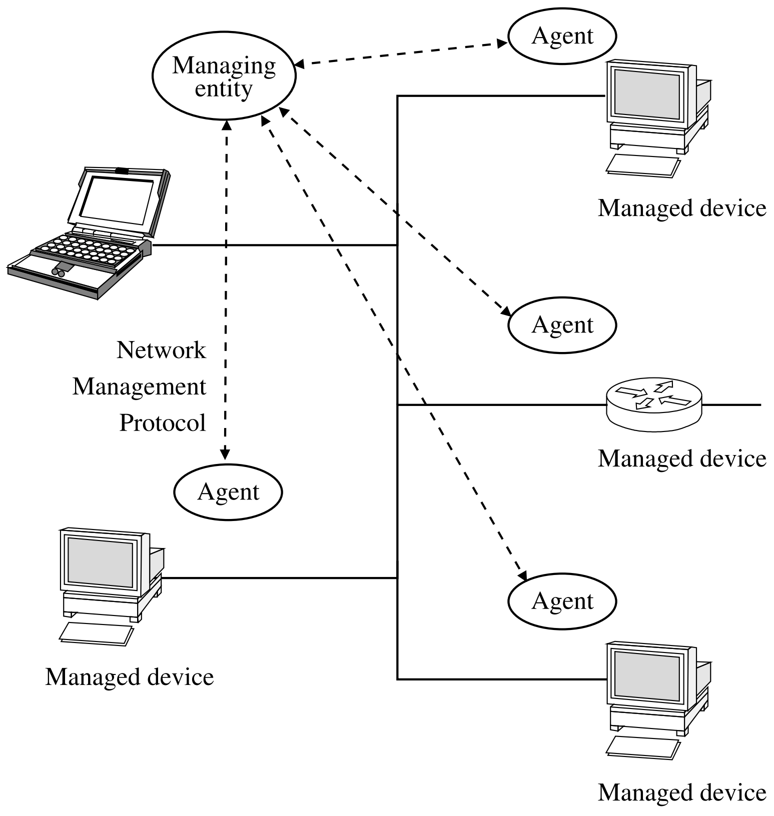


Figure 3.1: Network management infrastructure

Even if we exploit the fact, that the protocol header information is stored at the beginning of packets, the amount of data is enormous: an one hour trace of the first 64 bytes of each IP datagram would count for 18 Gbytes on a one Gigabit Ethernet link. Therefore packet monitoring is infeasible for any long-term storage.

The other way how to collect information about network state is the *Network Management Infrastructure*. Although its primary interest is managing network entities, several properties can be exploited for network measurement. As shown in Figure 3.1, an infrastructure is comprised of several principal components: The *managing entity* is an application that controls the process of collecting, analyzing and displaying the management information. It also provides user interface for the network administrator. A *managed device* is simply a node on the network that is being managed by the managing entity. The *network management protocol* determines communication protocol between the two mentioned entities. Finally, a process called *network management agent* runs at the managed device. It communicates with the managing entity and executes actions on its behalf.

Each managed entity collects statistics about its state and stores it in a virtual database, so called *Management Information Base* (MIB). MIB database is hierarchical and its objects are defined by the ASN.1 notation. It is comprised of several distinct MIB modules. MIB-II [25] defines the base for management of TCP/IP-based Internets and thus can be exploited for network measurement.

To communicate information from MIB to the managing entity, a Simple Network Management Protocol (SNMP) [5] may be used. The way for MIB objects transmission is via

`GetRequest` PDU, which gets a value of one or more object instances, or `GetNextRequest`, that gets a value of next object instance in a list or table.

MIB and SNMP provides a way for collection of coarse-grained statistics about the network state. Unfortunately, no specific information contained in the packets' network headers is retained and thus the usage is very limited for any extensive globally-scoped measurement.

A fair compromise between amount of data in packet measurement and information loss in MIBs is *flow based measurement*. Unlike MIB-II, where data are aggregated per interface (IP group for instance), in flow monitoring data are aggregated per flow. Since flow based measurement is one of the objects of this thesis, it will be further described in next chapter.

3.3 Flow based measurement

The most important term is, indeed, a *flow*. We will use the definition from [37]: A *flow* is defined as a set of packets passing an observation point in a network during a certain time interval. Packets of the same flow have common properties, which are defined as the result of applying a function to the values of: a) one or more header fields (for instance source IP address), b) one or more characteristics of the packet itself (number of MPLS labels, . . .), or c) one or more fields derived from packet treatment (for instance next hop IP address, . . .). This definition covers the range from a flow consisting of several packets to a flow consisting of just a single packet. The common properties which distinguish flows are referred to as *Flow Keys*.

A generic architecture for flow measurement requires several other terms to be defined. An *Observation Point* is a location in the network, where IP packets are observed. It might simply be an ingress interface of a network switching device, its mirrored port, tap, etc. Observed packets are processed by *Metering Process*, which includes several algorithms essential for:

- Packet header capturing and timestamping
- Application of sampling
- Flow lookup, update and maintenance

The product of metering process is a data structure, *Flow Record*, that carries information about IP flows.

Metering process might have a complicated architecture, with a mixture of various types of algorithms. Since it is a root of a measurement device, these algorithms will be analyzed in Chapter 4.

Exporting Process captures created flow records and sends them to one ore more *Collecting processes*. These are then stored for further processing.

The objects defined above constitute a flow measurement architecture. Further, exporting and collecting processes are hosted by devices, called *Exporter* and *Collector* respectively. An example of such architecture, based on monitoring probe, is in Figure 3.2. The router in the figure provides Internet connection to the local area network and a mirror connection for measurement device (a probe) that consists of metering and exporting process. Flow records are then transmitted to one or more collectors. A flow records exchange process between exporter and collector is held by a communication protocol (a collector is usually device remote to exporter). The two most used are Cisco *NetFlow* and *IPFIX*.

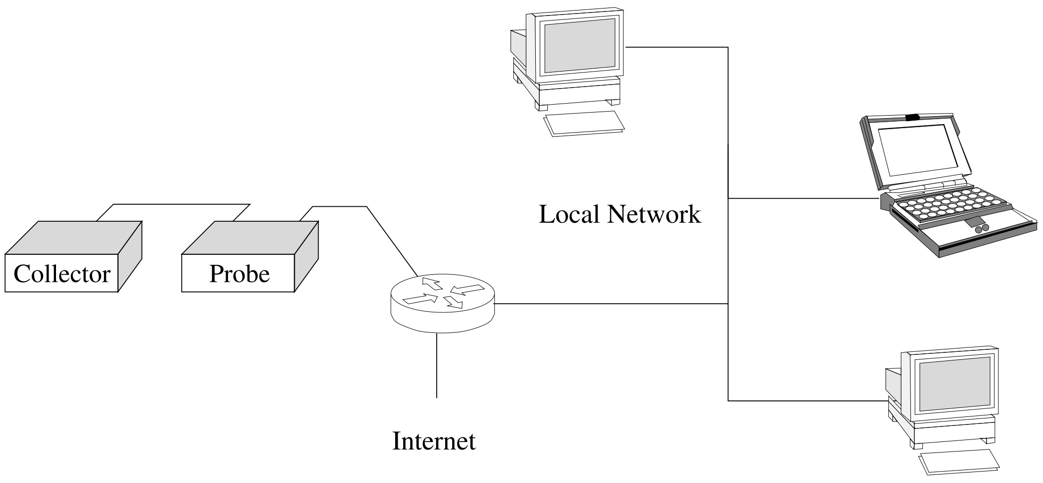


Figure 3.2: An example of flow measurement architecture.

3.3.1 NetFlow

Cisco NetFlow [6] is a protocol for IP flow exchange between exporter and collector. Although proprietary, it is open and thus not limited to Cisco devices. NetFlow itself defines the format of flow records encapsulation and the means of their transmission from exporter to collector. There are several version of the protocol, starting up from NetFlow version 1 to version 9. On top of that stands Flexible NetFlow, which defines a user configurable infrastructure for traffic measurement.

A flow in the NetFlow protocol is defined as a unidirectional set of packets sharing the following values:

- Source and destination IP address
- Source and destination UDP or TCP port
- IP protocol value
- Ingress interface
- IP type of service field

These values are referred to as *Flow Keys* and they together define an unique identifier for a flow record. The identifier, together with the aggregate data (total bytes, packets, etc.), comprise the flow record.

Netflow version 5 is the most widely used protocol variant for flow record transmission. The datagram format is depicted in Figure 3.3. The first part is the NetFlow header, specifying the version of the protocol (with additional system information data), followed by one or more data sets (in case of version 5 these are flow records). The actual record format is fixed and contains flow key values together with aggregated data.

NetFlow version 9 [6, 7] addresses flexibility issues with earlier protocol versions. The datagram format is not restricted by version 5 definitions, but is user definable. Thus, the basic format in Figure 3.3 has been extended to a more universal one (Figure 3.4). The header of the datagram is followed by one or more *FlowSets*. Each FlowSet has an identifier associated with it and it defines the type of data set. The category is one of the following:

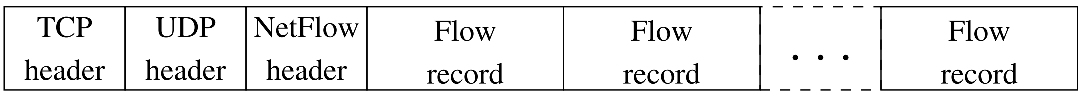


Figure 3.3: NetFlow version 5 datagram format.

- *Template FlowSet* defines the structure of the actual flow record transmitted from exporter. It is an essential part of NetFlow version 9, because it allows collector to process Flow Records without necessarily knowing the interpretation of all data in the flow record.
- *Data FlowSet* carries actual values of a Flow Record. The structure must have been defined by the appropriate Template FlowSet, before the data transmission has been initiated.
- *Options Template FlowSet* does not supply information about IP flows but rather information about the measurement process itself (for instance an interface sampling rate).

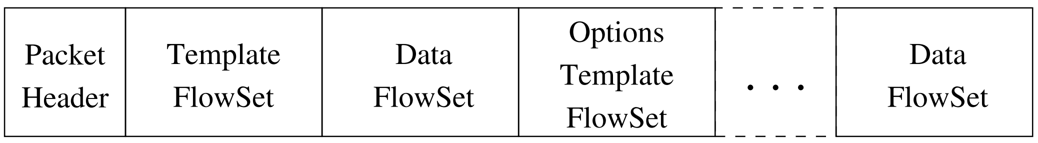


Figure 3.4: NetFlow version 9 datagram format

A single NetFlow version 9 datagram may contain all of the three mentioned FlowSet variants (Figure 3.4), or just a single chunk of Data FlowSets for flow records exchange.

3.3.2 IPFIX

Although NetFlow is open, it is a proprietary protocol. Thus, an effort is being made to create a standardized version for flow exchange. IP Flow Information Export (IPFIX) defines not only the communication protocol between exporter and collector, but also requirements on the whole monitoring infrastructure, starting with monitoring process and ending with collector. Although not yet standardized, the IPFIX Working group has created a set of documents describing requirements and information models for IP flow creation and export.

Requirements for IP Flow Information Export [37] document builds upon specification of possible applications requiring IPFIX. The requirements for the monitoring infrastructure are then derived to meet the selected criteria. These are the following:

- A metering process must be able to distinguish flows according to several criteria, that include interface number, IP header field, transport header fields, or MPLS labels.
- Requirements for the metering process. These include sampling ability, protection against resources exhaustion, proper packet timestamping and flow management
- Requirements for the exporting process. These include the specification of information model (e.g. what data are exported), data model (e.g. how data are represented in

flow records), means of data transfer from exporter to collector and other essential information.

- Requirements on configuration of metering and exporting process.

In [35] an information model for IPFIX is defined. All possible elements of the IPFIX protocol are described here. Information elements are grouped into several sections, with the most important pointed out here:

- IP header fields
- Transport header fields
- Sub-IP header fields. These may include link layer fields or fields between link and IP layer (e.g. MPLS).
- Per-Flow counters and Min/Max flow properties
- Timestamp information

Another essential document [8] deals with the specification of IPFIX protocol for data exchange. One of the key parts of the document is the PDU format. It is based upon NetFlow version 9. Because all necessary information has been written in Section 3.3.1, the IPFIX format will not be described here in detail. [8] determines transport layer protocols for the data exchange as well. SCTP [42] or TCP may be used for transmission over congestion-susceptible links. UDP may also be implemented.

3.3.3 Applications of flow measurement

When dealing with high-speed and large-scaled networks, traffic monitoring and management is essential for administrators or providers. Flow measurement provides useful insight into the state of the network and may be directly used for planning, detection of anomalies or may be a basis for further, higher level analysis of collected data. [36] and [12] define these basic applications of traffic flow measurement:

- *Usage-based accounting* is one of the key means for Internet Service Providers (ISPs) to charge customers for byte usage. User can be charged based on IP address or traffic type (higher-level protocols, TCP/UDP ports) and on time or volume of the traffic.
- *Traffic engineering* is a process of controlling how traffic flows through one's network in order to optimize resource utilization and network performance [47]. The key objectives might be minimization of packet loss and delay, maximization of throughput and uniform resource utilization [1]. One of the ways to simplify this task is to provide traffic information by flow measurement.
- Flow measurement may be used as a basis for *attacks* or *intrusion detection*. Anomalies on network, indicated by flows may show intrusive attempts for port scanning, denial of service attacks (DoS), distributed DoS, etc.
- Current TCP/IP model provides only limited means of *QoS* (IP ToS field). Using flows, traffic can be analyzed and differentiated per flow to ensure minimal requirements on delay, jitter or packet loss.

- For *Heavy hitters* or *Application and User profiling* flow measurement may be exploited. Monitoring dominant components in their network, administrators may plan new topologies or topology changes, determine the most used applications or investigate corporate policy or security violations.

Chapter 4

High-speed monitoring principles and algorithms

When dealing with small Local Area Networks (LANs) with rates below 1 Gbps, the time needed to process one packet is sufficiently large for software solutions. As the data is aggregated into backbone connections with rates far beyond 1 Gbps (which is common in current national or international networks), monitoring devices are much more resource-demanding. One way to overcome this growth is to deploy a scalable solution. It might be a set of distributed monitors that operate at the edges of the network instead at its core. However, this might not be the right solution, because requirements on maintenance and manageability of such architecture may degrade its advantages.

If distributed solution of simpler but slower monitors is not desirable or even not possible, when approaching 10 Gbps rates, parallel architectures must be exploited. This chapter deals with algorithms and principles for IP traffic flow measurement on high-speed networks. If we adopt terminology from Section 3.3, and use IPFIX IETF draft [38] for generic architecture description we can derive three most important components that make up measurement architecture:

- Metering process
- Exporting process
- Collecting process.

In this chapter we will focus on the metering process of such architecture and describe in detail what algorithms and techniques may be used to achieve best performance at reasonable cost of resources. Since we want to focus on parallel processing, emphasis will be given on hardware implementations of such algorithms. Note that exporter and collector side, i.e. the process of capturing flow records from the metering process, their wrapping into specified export protocol, followed-up by their storage and analysis, are beyond the scope of this document and are not discussed here.

Figure 4.1 shows a generic configuration for a metering process. When packet enters the system it is firstly preprocessed and necessary information is extracted that is needed to lookup flow that this packet belongs to. This is dealt with in Section 4.1. Extracted packet headers then enter the component that finds corresponding flow in flow cache, updates it with the newly arrived information and stores back to memory (section 4.2). Concurrently, another process runs, that maintains the state of all flow records held in memory (Section 4.4).

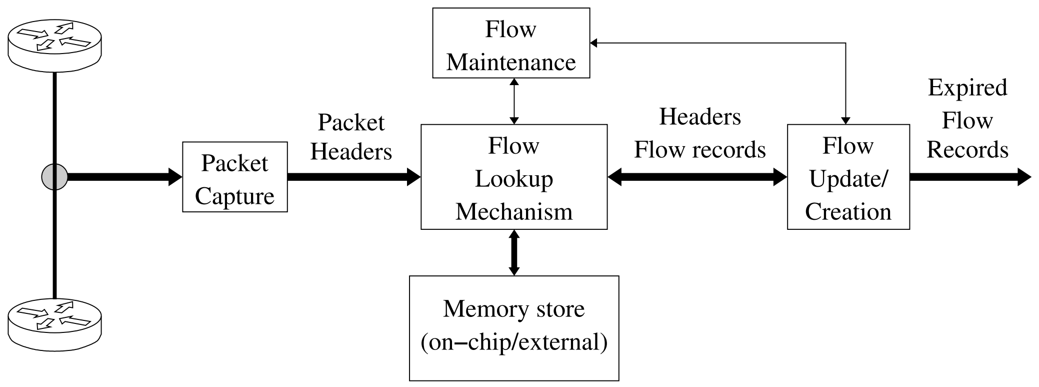


Figure 4.1: Generic metering process architecture.

4.1 Packet capturing

Packets entering the system via ingress interface are at first preprocessed. Useful properties might be derived, such as interface number, packet length, timestamp and added to the packet header. Moreover, for measurement, not whole packets need be processed. Instead, protocol header fields and/or part of the payload could be extracted and sent for further processing.

Another aspect is the reduction of the amount of data entering the system, performed by *input sampling* or *filtering*. Filtering might be useful when a specific type of traffic is of user's interest. If it is known, rules may be set and a filter preprocessor created to reduce traffic load. Since filtering is not the primary interest of flow measurement, we will rather describe several sampling techniques in detail.

4.1.1 Sampling

When it is not possible to use filtering, because the type of the data of interest is not known, sampling may be used. This is especially true in network measurement, because traffic mix is unknown and variable. There are two main reasons for sampling to be employed:

- A monitoring device cannot properly handle the worst case scenario, when an overwhelming amount of data enters the system. In this case, instead of packet loss, which cannot be properly controlled, rather a controlled mechanism is desirable, where packet rate is systematically lowered by sampling and sampling parameters can be reported by the metering process.
- The other reason is protection. Let's consider a simple example: DoS attack. Each incoming packet creates new flow record that occupies space in flow cache. Device resources may quickly get exhausted, therefore there is a need to reduce amount of packets entering monitoring system. This may be accomplished by changing properties of flow definition, but if it is not possible, sampling must come into play. Note that immediate flow expiration would not help in this case, because exporter and collector side would still be overloaded.

When defining sampling parameters, a *trigger* mechanism must be selected that determines how objects are selected for processing [12]. Count-driven triggers use an increasing sequence of counts $i_n : n = 0, 1, 2, \cdots$, where i_n denotes object that is sampled. Conversely time-driven triggers use sequence of times $\tau_n : n = 0, 1, 2, \cdots$, each τ_n denoting time at which a sample is accepted. i_n and τ_n are defined by parameters of the sampling process. These can be defined in several ways, we will describe here the simplest ones:

- *Deterministic* sampling is the simplest form, where measurements are separated by a fixed interval of time[1], defining the sampling ratio. An example is in Figure 4.2. Sampling rate is 1/3 and therefore packets $0, 3, 6, 9, \ldots$ will be accepted. Although this type is very simple to implement, one of its serious drawbacks is periodicity. If the traffic observes periodic behavior with period close to that of sampling process, there is a possibility that this behavior will be only partially observed by monitoring process. Again, an example is shown in Figure 4.2. The sampling ratio is 1/3. Selected packets are marked with arrows, other packets are discarded. The shaded ones, for instance, may represent a malicious traffic, which in ideal case won't be observed at all. This situation is probably not possible in real traffic mixes, but if sampling rate is very small (1/100 for example) malicious behavior could be partially hidden.

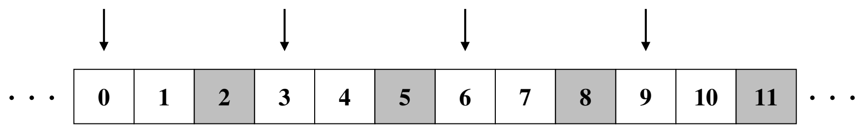


Figure 4.2: Deterministic sampling with period 1:3.

- *Uniform random sampling*[2] is another fairly straightforward technique. Each packet is sampled with probability $p = 1/N$ determined by sampling rate (Figure 4.3). An implementation is simple: use an uniform random number generator with range $< 0, 1 >$ to generate a value N. If $N \leq p$, accept incoming packet, otherwise discard it. Hardware implementation would be similar, except that integer values could be used.

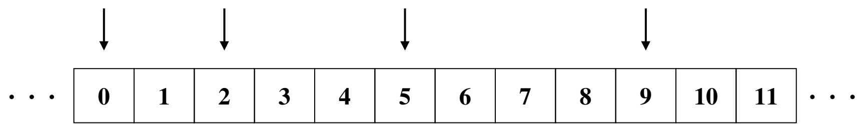


Figure 4.3: Uniform random sampling with period 1:3.

Random sampling has better properties than deterministic, because it is not predictable in advance and is recommended in cases if it bring advantage over some more sophisticated types [29]. It is thus desirable to use it in monitoring devices, because of fairly simple implementation and good statistical properties.

4.2 Flow lookup

After the packet has been preprocessed, a corresponding flow must be picked up and updated. If the flow does not exist yet, it must be created. As described in Chapter 3.3 a

[1]Time in this case can as well be in means of packets.
[2]Also referred to as Geometric Sampling. See [29] for more details.

flow is defined by properties that are shared among a set of monitored packets, termed flow keys. These can be one of the following:

- packet header fields
- properties of a packet itself (e.g. number of MPLS labels, etc.)
- field derived from packet treatment (e.g. next hop IP address, ...)

Key fields uniquely identify a record and a function that maps them to an actual flow record pointer value must be identified.

4.2.1 Naive indexing

The simplest case, when a flow identifier (flow ID) directly maps to flow record pointer is not feasible, because an enormous amount of reserved memory must have been used. For example, and IPv6 NetFlow record key fields might be over 300 bits long. Masking out a portion of the raw identifier might not be desirable as well, because such an identifier produces a lot of collisions (e.g. two different flows have the same identifier). Moreover, in such implementations, possible attacker could exploit reduced variability of flow IDs to modify the traffic in order to attack monitoring device. Therefore flow lookup must be more sophisticated and concurrently preserve the simplicity of direct addressing.

One option is to implement lookup based on hash tables. We will present and compare three types: the simplest hash table, called here *simple naive hash table* (SNHT), then *naive hash table* (NHT) and *fast hash table* (FHT, Section 4.2.2). The two latter terms are adopted from [41].

The realization of SNHT would be based on a table of flow records T and a hash function h (Figure 4.4). If a raw flow identifier is presented, a corresponding flow f is picked up from memory, i.e. $f = T(h(x))$. Three possibilities may happen:

1. A flow is not valid and thus new item must be created
2. A flow is valid but its key fields do not match the packet ones, i.e. collision occurred. Received packet must either be discarded, or the current flow replaced by a new one.
3. A flow is valid and its key fields match the packet ones. The flow is then updated.

In NHT, $h(x)$ is a pointer to the list of items (figure 4.5). Given $h(x)$ for a particular x, the list is sequentially searched to make the definite lookup decision. In case of match the flow is updated, otherwise new item must be created.

Both cases, however are not generally collision free, because when a map $h : A \rightarrow B$, where $|A| > |B|$, is applied, at least two elements from A must map onto the same element in B. Even if the collision probability could be kept low when using whole hash output space[3], it would require several Gigabytes of memory to implement such functions.

4.2.2 Enhanced indexing algorithm

Another hash-table based algorithm might be used in flow lookup, that on average is faster than NHT. We will refer to it as a *Fast Hash Table* (FHT) [41].

[3]In [26] for instance, authors use MMH hash function [16] with low collision probability if its 32 bit output is used as an index.

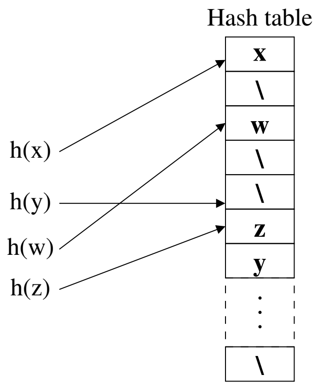


Figure 4.4: Direct hash indexing.

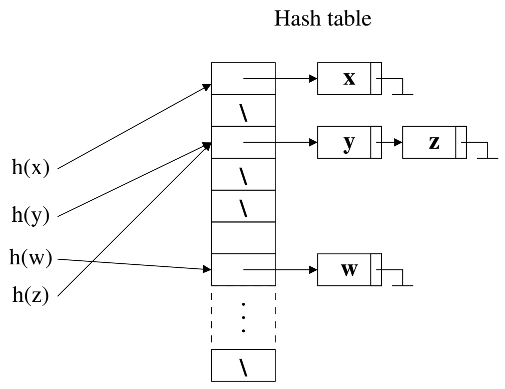


Figure 4.5: Hash table based indexing.

Firstly, *Bloom filter* must be described, which forms the core of FHT. Bloom filter is a hash-based data structure to store a set of items compactly. Given an item x it computes k hash functions: $h_1(x), h_2(x), \cdots h_k(x)$. These functions are an address into bitmap of size m. When inserting and item into the structure, all k bits, computed from x, are set to 1 (assuming that the bitmap has been set to zero before first insertion). An item lookup is performed in similar manner: For x, $h_1(x)$ through $h_k(x)$ are computed and k bits picked out from the bitmap. If all of them are set, an item is present in the structure. A minor drawback is that false positives may occur. But if unique identification is stored with an item (such as that stored in flow record), false positives can be detected and the item rejected in lookup process.

A more important drawback of this structure is that items cannot be removed. Therefore in [13] *Counting Bloom filter* has been proposed. The bitmap in basic structure is replaced with a set of counters. Each time an item is added into the structure, all of k counters addressed by hash functions are incremented. Deletion of an item is the reverse operation. Again, an item is present, if all k addressed counters are non-zero.

We can now proceed to describe FHT. An array of m counters is maintained, where each counter is associated with a bucket in the hash-table (bucket is composed of a list of items). Insertion procedure computes k hash functions over an input item and increments all k counters indexed by the hash values. Then, the item (if it is not in the table yet) is stored k times into lists associated with the indexed counters. The insertion operation is illustrated in Figure 4.6. Four items, x, y, z, w were inserted. Each $k = 3$ times.

The speedup of the algorithm over NHT comes from the search operation. After $h_1(x), h_2(x), \cdots h_k(x)$ have been computed the algorithm has k counter values, that determine size of the list for buckets associated with them. If all counters are non-zero, bucket associated with the counter with smallest value is sequentially traversed to find the searched item. If the hash table is stored in an off-chip (and hence slow) memory, the time to traverse the list may be crucial for the throughput of this algorithm. In [41] it has been shown that probability of a list in the FHT being filled to j items is much less than in classic NHT algorithm and thus on average the list-search time is shorter.

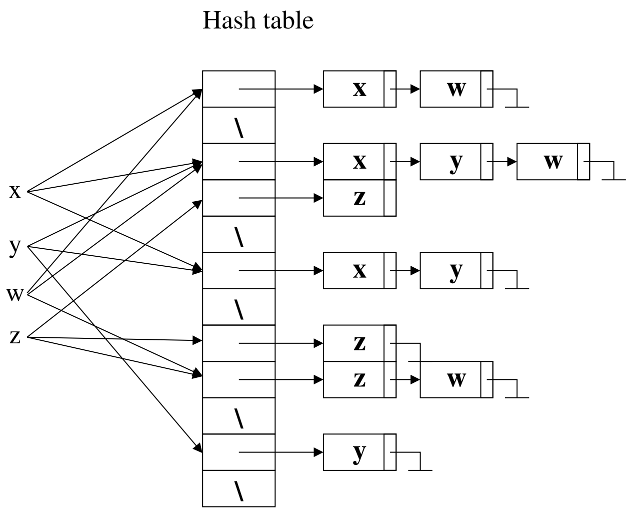


Figure 4.6: Fast Hash Table insertion procedure.

4.2.3 Comparison

All three methods may be used depending on circumstances of specific application. The SNHT case is the easiest one to implement, because it requires only one hash function and its result is simply a pointer to the memory of flows. The time to access an item in the table might be less than in NHT, because a bucket always contains only one item (or none). However, the SNHT solution will produce more collisions. The FHT algorithm is more efficient than NHT in terms of access time, but its one serious drawback is memory requirements. The space needed to store n items would be kn with k hash functions. Thus, careful evaluation of conditions must be done to choose a proper algorithm for flow lookup.

4.3 Flow update

After the lookup operation is finished flow update starts. If the requested flow is present in flow cache, information in packet header (e.g. packet and byte counts, TCP flags, flow end-timestamp, etc.) are aggregated into the flow. It is then stored back to memory. During the update, several conditions must be resolved:

- During aggregation, one or more fields in flow record might overflow. Proper action should be defined, either by a user or in advance at design phase, what action will be applied in such situation.
- In some cases a flow record state must be checked for certain conditions (for instance to check active timeout, Section 4.4) and if they are met a proper action taken. One of the actions might be to release the record from flow cache and export it to a collector.
- If the flow lookup algorithm does not resolve collisions, flow update component *must* be able to detect when two flows map onto same data item in flow cache. Otherwise monitoring will get disrupted.

If the lookup procedure did not find a flow that belongs to the processed packet, new flow must be created and filled with flow keys and initial aggregate data.

Again, the flow update procedure must be as fast as possible to meet the packet time budget, so it might be feasible to be hardware-implemented.

4.4 Flow maintenance

Beside processing discussed in previous sections another process must be implemented that runs concurrently and maintains the state of flow records stored in memory. It periodically checks for flows that either do not observe packets for a long time or flows that simply last too long. If it detects such flows, it must release them in order to make space for the new ones. If such maintenance procedure would not be implemented in monitoring system, these flows would simply stay in flow cache until the device would be stopped or at least until they would be replaced by new ones.

Following situations can be recognized when dealing with flow activity:

- No packets have been observed for a flow for a specified time interval, called *inactive timeout*. After the timeout has expired for a particular flow, the system should release and report it to the collector side for further analysis.
- To avoid ever-lasting flows, *active timeout* is used, which is in contrast with the previous one. If the flow record is active for a time interval defined by active timeout, a monitoring device is supposed to report it to collector.
- Flows may also be expired when the device has not enough resources to store new flow records [38].
- Another situation may occur, if the flow is terminated with an *explicit notice*. For instance TCP flows may be terminated by the FIN control bit.

It is straightforward to implement active timeout checking if each packet entering the system carries its (unique) timestamp. Then the flow record could contain the timestamp of the first packet (start of the flow) and with each packet arriving, its timestamp would be compared to the one stored in the record. All other "activity enforced" checks can be performed this way, because they can simply be included in the flow update process.

In the case of active timeout, the extra information present in the flow is not so much redundant, because it is used to determine the start and duration of the flow. However, to check for inactivity of flows, a periodic activity must run in the background and store information about the flows' state as well. Such process must consume extra resources (of the chosen platform): extra memory to store the state of the flows and extra processor cycles to perform the periodic inactivity checks. In any case, implementing the timeout mechanism is crucial for a correct monitoring device's functionality and so cannot be neglected.

Following [26], we can describe several types of inactive-flows selection heuristics. One of the most efficient is the *Least Recently Used (LRU)* strategy. This technique can efficiently be implemented using a double-linked list. Each item in the list holds a pointer to a flow record in the memory. For each packet that arrives into the system and has been classified, i.e. the flow pointer has been determined, the list is accessed and the item rebounded into the beginning of the list. Thus, the most recently updated items are at the start of the list, whereas the oldest ones are in its tail. Determining timed-out flows simply requires scanning the list from its end and comparing the last update time with the current time (assuming that the flow record contains the timestamp of the last packet or other similar data). The LRU implementation requires additional data to be stored for each record. Namely, two list pointers and the timestamp of last update. If all these components are four bytes in size, and the system capacity is N flow records, an overhead of $12N$ bytes must be accounted for. If we estimate the necessary cache capacity of the system to half million records, the overhead of the double-linked list implementation is approximately 6 Megabytes. This amount of data is probably not significant in software implementations. But as can be seen later in the target platform description (Section 5.4), for hardware implementations, where the capacities of memories are substantially limited, keeping memory requirements low may affect the system significantly both in its cost and its throughput.

The LRU algorithm requires a kind of sorting which is realized as a double link list. Another approach is *cyclic check*. Every item in the monitoring system has a timestamp of its last update stored with it. The check periodically proceeds through the items, and checks for inactivity of the item. If it reaches certain, user defined level, the item is removed from the memory.

Chapter 5

Analysis

In order to determine requirements on throughput of a hardware accelerated monitoring system, this chapter presents results of several simulation experiments. Two aspects were considered: profiling of a commercial, highly optimized software monitoring probe and aggregation factor of basic indexing algorithms described in Chapter 4. Both factors are very hard, if not impossible to derive analytically, therefore thorough software simulations have been carried out to estimate them. The description and results are presented in Sections 5.1 and 5.2.

As part of the analysis, Section 5.3 brings discussion on the variability of the resulting system. The monitoring device must be designed in such a way that a user be allowed to change the monitoring process as quickly and as flexible as possible. The section introduces several possible applications which require flexible monitoring process.

The target platform with its characteristics and limitations will be introduced as part of this chapter as well. Although the system should be targeted at this platform, the whole system is fairly hardware independent and therefore should be easily ported to different hardware acceleration devices. The platform is described in Section 5.4. Moreover the requirements on the hardware accelerated process are addressed in Section 5.5.

5.1 SW probe profiling

This section describes an analysis of a commercial software flow monitoring probe in order to estimate the maximal packet rate this probe is capable to process. The results presented here can be extrapolated to a characteristics of a *secondary flow aggregator* engine used as the software part of the monitoring application (see Figure 5.6). From the results, the minimum aggregation factor can be derived as well and this value may serve as an indicator of how much the hardware part of the system must be tuned in order to process the desired ingress data flow without packet loss.

The probe used in the profiling was a highly optimized application developed by the INVEA-TECH company [17]. The application follows the generic flow monitoring system architecture defined in Chapter 4 and its basic partitioning is in Figure 5.1. When a packet enters network interface, it is copied to main memory and then handled by the application. Note that the kernel processing is bypassed in this configuration in order to improve the incoming packet rate. The packet is then preprocessed by the *Header Extraction* part, which extracts desired protocol header fields. The flow key fields are then hashed to form a unique flow identifier. The rest is an aggregation mechanism that handles (i) flow lookup,

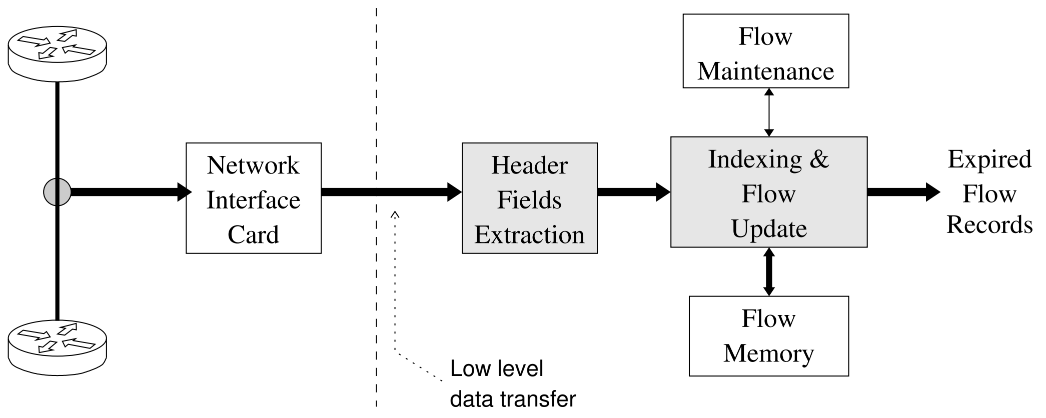
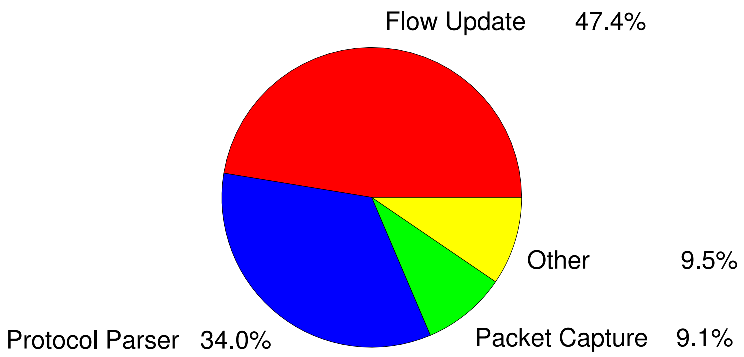


Figure 5.1: INVEA-TECH Flow monitoring application.

(ii) flow update and (iii) flow maintenance. The last three parts are merged into one in this configuration. The aim is to gather profiling data for each part in order to estimate the limits of such system.

A local university campus was monitored for the measurement. The link was loaded on average with 4.1 Gb/s and 668,000 packets/s. From the performed measurements the maximal packet rate the system is able to process can be derived. However, the measured program contains profiling information which poses a significant penalty on the system performance. Therefore, the CPU load measurement has been done for a non-profiled version as well. Figure 5.2 summarizes the profiling information. It shows that for this packet rate, the profiled version loaded the CPU at 32 %, while the non-profiled version caused more than four times less load. The load difference is the profiling overhead of the system. We will therefore use the non-profiled version to derive the maximal packet rate one can reach using the measured application.



Pac/s	bits/s	Capture	Protocol Proc.	Flow update	CPU load	
					Profiled	Non-profiled
668,211	4.1 Gb/s	9.1 %	34 %	47.4 %	32 %	10 %

Figure 5.2: Software probe load on Intel(R) Xeon(R) 3075 (64b), 2.66GHz, 4096 KB L2 cache, 2048 MB RAM.

If the system load would scale linearly, with 668,211 packets per second having 10 % CPU load, the packet rate would scale to 6.6 million packets per second. However such assumption cannot be made, because CPU load generally depends on many factors and is mainly limited by an on-chip cache. Also, the number of flows per second entering the system increases with increasing interface speeds and therefore might cause more severe cache misses during software processing.

To properly estimate the rate of CPU load with respect to input throughput would require measurements based on higher packet rate than in the traffic used for profiling. However such live packet channel was not available and using synthetic traffic generator is not appropriate for this task[1]. Thus the maximum packet rate will be only roughly estimated to the half of linearly extrapolated value, 3.3 million packets per second.

In the next, it is assumed that the monitoring process consists of a two layered system. The data is firstly preaggregated in hardware and flow records are then exported and processed by the software layer. The results gathered from the software probe profiling from Figure 5.2 show that the protocol parsing part consumes around 34 % of CPU load. However, When the software system works as a secondary aggregator only, this can be eliminated. Therefore, the software aggregator performance can be approximated to 3.3/0.67 million packets per second which is around 5 million packets per second.

The estimated 5 million packets per second can be doubled when using two CPUs, when two independent L2 caches are considered. Thus, with such configuration, an estimation of 10 million packets per second can be reached for software solutions.

5.2 Real traffic simulations

The goal of a flow monitoring system is to achieve maximum packet aggregation with respect to a chosen aggregation scheme, i.e. selection of flow key fields. The limitations on an aggregation scheme may by crucial, because even if the hardware probe would be able to process high rate data without packet loss, if it is designed poorly, it might overwhelm the software part (secondary aggregator, collector or other analyzer).

It is supposed there are three most influential factors with respect to the aggregation achieved:

1. Flow record definition, which can be defined in a lot of variations and is totally in the hands of user. It is the most influential part but unfortunately cannot be predicted by the designer. The flow record type defines the coarseness of information received by the user. The more fine-grained it is, the more load is enforced upon the monitoring device.

2. Size of the flow cache. In real time traffic, there are normally several tens of thousands of flows per second, which fill the average flow cache in less than few seconds. Therefore, the larger flow cache, the bigger aggregation factor can be achieved.

3. Flow indexing type, which might be either a simple direct-hash addressing, an index sequential hash table or any other more sophisticated approach. However it should be noted, that the more the approach is refined, usually the bigger resources it consumes, mainly in terms of on-chip memories or computation resources.

[1]A synthetic traffic generator should reflect specific traffic mix, packet length, burstiness and flow distribution of live traffic. It is therefore very difficult to simulate such behavior.

Sample	Packets	Bytes	Flows	Aggregation	Duration
10 Gb/s	197 M	161,533 M	11,366,428	17.36	120 s
Per second	1.644 M/s	10.768 Gb/s	94,720	N/A	N/A
20 Gb/s	386 M	305,976 M	22,606,386	17.09	120 s
Per second	3.219 M/s	20.399 Gb/s	188,306	N/A	N/A

Table 5.1: Trace used for offline analysis.

The aim of this section is to analyze live traffic traces with respect to properties (2) and (3). The traces were obtained from the CAIDA organization [4].

5.2.1 Packet traces

We will start with the traces description. Two use cases were defined to test the two selected indexing algorithms: one two minute 10 Gb/s and one sixty second 20 Gb/s fully loaded link. Because the OC192 CAIDA links were not fully utilized, several independent traces were merged into one, to form the desired data rate.

The traces statistics are arranged in Table 5.1. Firstly, let's have a closer look at the amount of packets and the amount of aggregated flows in both samples. We shall define the flow as a five-tuple: source and destination IP address, source and destination transport port, and transport layer protocol. Using this definition the maximum aggregation, as defined by Equation 5.1 is approximately 17 for both samples. This is the packet aggregation limit that can be achieved if the flow cache would be sufficiently large to hold all flows in the sample. Also it should be noted that the average packet length in these samples is approximately 700 bytes. Therefore, the maximum byte aggregation[2] is much larger for live traffic.

In the measurements, however, it is assumed that the average packet length is much shorter. Then the aggregation factor defines the worst case. If the hardware accelerated part of monitoring system is able to preprocess data in order to meet the software limitations, no packet loss will occur. Finding out how much aggregation can be achieved with respect to the maximum aggregation factor, given by the trace characteristics, is the aim of the next two subsections.

5.2.2 Direct hash indexing

The scheme from Section 4.2 will be considered as direct hash indexing case. The hash computed from flow fields is simply used as an address into the flow record memory. The flow identifier is stored together with the record to identify possible collisions. Such scheme does not require any overhead except for the flow identifier that occupies extra space in flow cache.

For this type of indexing, there was only one experiment parameter, size of flow record memory. The range was chosen with respect to the flow record size, which was defined to be 64 bytes or 32 bytes. With 32 bytes, the maximum amount of flow records kept in memory is 524,288. The simulation results are presented in Table 5.2.

The aggregation factor in the Table has been computed as the total number of packets in the trace, divided by the number of collisions. Because the flow cache capacity is far

[2]Total amount of bytes divided by total number of bytes in a flow record.

smaller than the overall number of flows in the sample, the flows left in the cache were neglected and therefore were not included in the result.

From the Table, first thing that may be noticed is that direct hash indexing does not aggregate traffic very well and causes a lot of collisions. It is therefore necessary to post-process the collided flows in a much larger cache in software. For our 20 Gb/s aggregated CAIDA traffic and flow cache capacity for 256,000 flow records, the aggregation factor is approximately 3.722.

	10 Gb/s		20 Gb/s	
Memory size [flows]	**Collisions**	**Aggregation**	**Collisions**	**Aggregation**
128 K	54,376,648	3.628	132,472,541	2.916
256 K	41,028,431	4.809	103,810,544	3.722
512 K	30,228,336	6.527	77,936,084	4.957
1024 K	22,154,411	8.906	57,475,216	6.722

Table 5.2: Direct hash scheme aggregation rate for a sample traffic.

5.2.3 Hash table indexing

The direct indexing scheme is very simple, but produces a lot of collisions. Therefore, simulations with an index-sequential algorithm were also carried out in order to assess how a proper hash table behaves in a flow monitoring situation.

The hash table is defined as in Section 4.5. We won't consider here a Bloom filter based hash table, because its implementation is too difficult to be implemented in hardware. The table is arranged into buckets, each bucket containing a linked list of flow records. A bucket is addressed directly, whereas a list is scanned sequentially, or in some circumstances, can be also scanned associatively.

Two table subtypes are defined in the simulations, according to the victim selection policy, i.e. when a bucket reaches its maximum list capacity, and a new item is to be inserted into the table, a victim must be selected: a LRU policy and a random selection. These two are compared.

Furthermore, a maximum list size might be selected. Several experiments have been done with respect to the maximal list size. The results presented here are for a list size of 8 items (flow records).

From the results in Table 5.3, it immediately follows that random victim selection policy is comparable to LRU policy. Surprisingly, in this case the random selection performs better than LRU. Because LRU in this context does not bring any aggregation improvement, we can conclude, that it would not be the right solution for this task.

What is more important, is that reasonable aggregation factors are emerging only with much bigger flow cache capacities than COMBOv2 platform offers (with a 64 byte context it is 256 K flow records). Reasonable aggregation factors are starting at approximately a cache for million flow records. However, with smaller cache sizes, the table still provides good aggregation.

Moreover, when comparing direct addressing with hash table at cache capacity of 256 K records, we conclude that with this policy, hash table does not offer significant improvements.

	LRU		Random	
Memory size [flows]	**Collisions**	**Aggregation**	**Collisions**	**Aggregation**
128 K	114330424	3.379	109526729	3.527
256 K	81434376	4.744	77009140	5.017
512 K	51489245	7.503	49840992	7.751
1024 K	35386052	10.92	34754818	11.12
2048 K	27144750	N/A	26988569	N/A
4096 K	21378229	N/A	21432908	N/A
8192 K	15267378	N/A	15339458	N/A

Table 5.3: Hash table aggregation rate for a 20 Gb/s traffic sample. The first column is the total memory capacity. List size has been set to maximum 8 items. Note that starting from 2048K flow memory capacity, no aggregation is in the table. The memory capacity was too large to estimate the aggregation factor from this sample.

5.3 Variability

In this section, flexibility aspects of a monitoring probe will be discussed. Knowing how much such system needs to be configurable is very important for proper system design and implementation.

Several export protocols have been described in Section 3.3, defining what data the monitoring system is required to extract from packet headers and what statistical data are gathered for each flow existent in the network. While Netflow version 5 has a predefined structure and it cannot be changed, it provides only limited means of monitoring, measuring number of packets and bytes accumulated during the lifetime of a flow.

Although Netflow version 5 protocol is the most suitable for e.g. usage-based accounting, detecting heavy hitters and other similar application, some monitoring applications require slightly different flow record structure. IPFIX draft thus defines a more extensible flow definition, derived from Netflow version 9. Here, the flow is defined in a same way as in Section 3.3 so the flow key fields do not strictly have to be IP addresses or transport layer ports as in the classical definition of Netflow. RFC 5102 [35] describes an information model for IPFIX, i.e. definitions of elements which may be put into a flow record. It defines a wide variety of packet header elements, from subIP, through IP, to transport header fields. Statistical flow properties, minimum or maximum or per flow counters can also be utilized to form a flow. Thus, the notion of flow might not strictly be limited to Netflow version 5 description.

Short name	**Long name**
mean_IAT	Mean value of packet inter-arrival time
var_IAT	Variance of packet inter-arrival time
var_data_ip	Variance of packet IP length
SYN_pkts_sent	Number of SYN packets sent
FIN_pkts_sent	Number of SYN packets sent
max_segm_size	Maximum segments size for connection

Table 5.4: Some of the discriminators that are usable for flow-based classification. Full list can be found in [27].

Another vote for flexibility is network traffic classification. Moore in [27] defines 248 discriminators usable in flow-based classification. Table 5.4 shows a sample of them. Most of the classificators exploit byte or packet counts and a variety of their mean and variance values. Another useful property might be the mean or variance of inter-packet arrival times for flows, which might carry information about the traffic dynamics. These discriminators have been successfully utilized in [28] for Internet traffic classification using Bayesian techniques.

In [14] a port-based technique using maximum entropy estimation has been developed in order to detect network anomalies. The traffic is classified by port numbers into several classes and for each of these classes a base distribution (acquired off-line) is compared to an on-line distribution to analyze for anomaly.

Several other approaches require flow-based statistics in order to classify traffic by size, duration, burstiness and rate. An example of a five-tuple approach is in [19]. In [21] flows are characterized according the four aforementioned properties. While size, duration and rate can be obtained from simple per-flow packet counters, to classify bursty flows, an inter-packet arrival time must be stored (defined as *train burstiness* in the paper). Inter-packet arrival time is a special property that is not considered in IPFIX.

To sum up this short survey, many applications require specialized information which is not included in Netflow version 5 flow record definition. Several other examples can be found that exploit many IPFIX definitions and which require information beyond IPFIX to be included in flow record (e.g. inter-packet arrival time). It can therefore be suggested that a flow monitoring device should be able to aggregate any user-defined information by a generic algorithm, possibly being user-customized as well. It is thus necessary to design the hardware accelerated monitoring probe as a flexible device which can benefit from the advantages of underlying hardware.

5.4 Target platform

5.4.1 Hardware

The main platform for this design is the family of COMBOv2 cards developed by the Liberouter project [23]. COMBOv2 cards family consists of a mother card, used as the data processing part and a so called interface card, which defines type of network connection. The are two variants, depending on the interface card:

- COMBOI-10G2 interface card with two 10 Gb XFP cages. This is the main platform for the probe architecture.

- COMBOI-1G4 interface card with four 1 Gb/s SFP cages used for Gigabit Ethernet connection. Although the design is primarily targeted for 10 Gb/s links (or more), this type of network connection may also be used.

The COMBO-LXT mother card (Figure 5.3) contains hardware components necessary for high speed network processing. Because these components define the limits of what hardware accelerator is capable of, they will be briefly described in the following text. The card consists of:

- Powerful Xilinx Virtex5 FPGA (starting with LXT50T and up to LX155T)

- Four Low Speed Connectors with throuthput up to 8 Gb/s

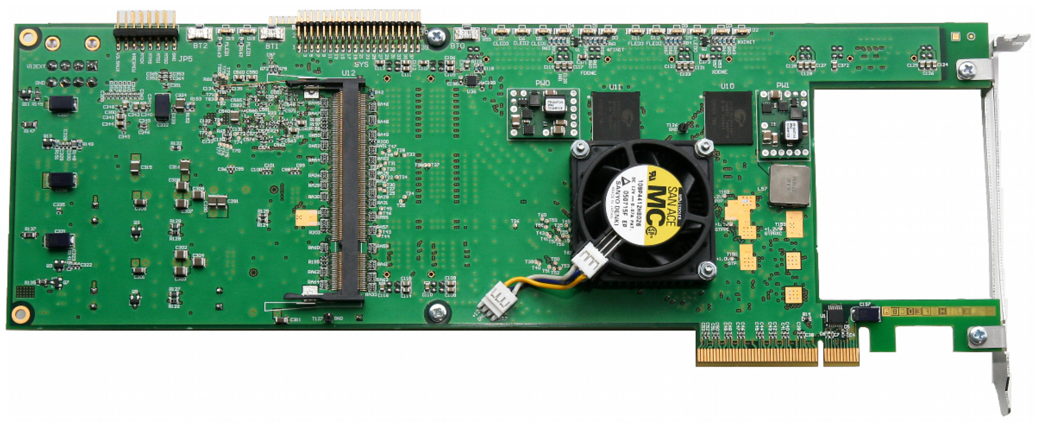


Figure 5.3: COMBO-LXT mother card without an interface card.

- Two High Speed Connectors with throughput up to 28 Gb/s
- Two QDR-II memories with full-duplex throughput of 8 Gb/s and 8 MB capacity. When using both memories in parallel total data throughput of 16 Gb/s can be achieved.
- SODIMM connector for DDR2 memory. Optionally, the card can be equipped with low latency high-speed RLDRAM memories with capacity up to 844 Mbits.
- 8-lane PCI-Express connector which provides throughput to the software of up to 16 Gb/s.

5.4.2 Hardware abstraction layer

In addition to the COMBO hardware the so called NetCOPE platform [24] is available. It offers an abstraction layer for the developer of high-intensive network applications. It also ensures at least partial portability of such applications. These can be then independent of network interfaces or the ways of data transfer into software part of the application. The platform is intended to fulfill the needs for quick application prototyping for a hardware-software co-design developer.

The platform follows a layered architecture and consists of several parts (Figures 5.4 and 5.5). The *hardware part* includes a hardware abstraction layer. With network applications as its main target it provides an unified network interface to a user. The part responsible for this is the **I/O Blocks** part. It parses incoming data stream from the interfaces and passes packets to an **application core** for further processing. After the processing stage is done, the resultant data is either passed to the software layer, by means of **fast DMA** transfer, or back to the I/O block to enter the egress interface. The application core is user programmed and it should be noted that it is completely unaware of the machinery behind Input/Output or DMA transmission.

Figure 5.5 shows the layered structure of the platform. Incoming data enters software through a PCI (or any other supported) bus and is handled by the kernel driver. It then passes data to an actual **software application** that is responsible for further processing. The transfer process between FPGA and back is therefore fully transparent. Only platform-defined protocols and interfaces must followed to keep the system working.

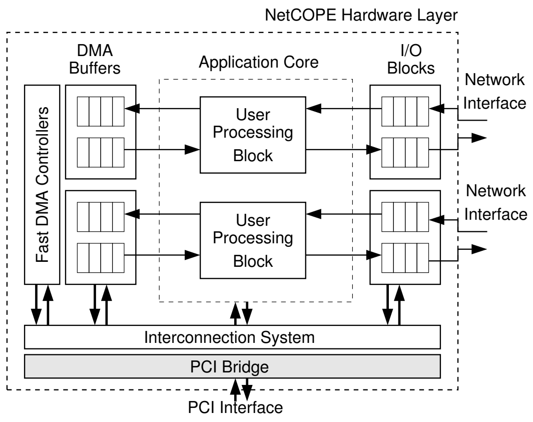


Figure 5.4: NetCOPE block structure.

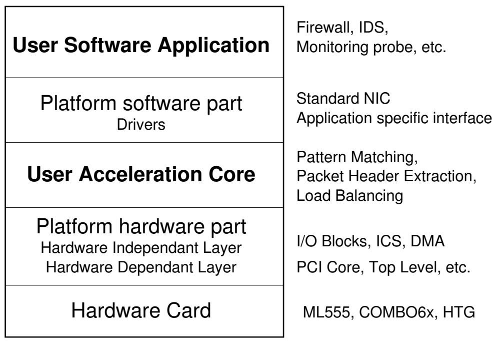


Figure 5.5: NetCOPE layers.

5.5 Processing throughput requirements

The hardware accelerator card cannot be used as a standalone application in most cases; it must be plugged into a PC. Therefore it is natural to divide the metering process into two independent sub-processes. The hardware accelerated part preaggregates data from ingress interface and creates primary flow records. These are not directly exported but rather processed by the software layer. Figure 5.6 shows brief outline of such architecture.

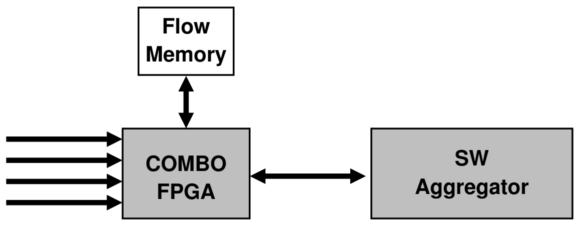


Figure 5.6: Implementation-independent flow measurement.

Note that it fits to our target platform introduced in section 5.4. The FPGA part is the NetCOPE application core and software aggregator is the NetCOPE software application.

Let's now define parameters of this generic system. Firstly, it should be noted that it is a data flow architecture. The data enters the system at an ingress interface, enters the hardware processing part where it is processed flows through a PCI bus where it enters the software processing part. Then it is exported to a remote collector or directly processed by the host computer. The hardware part also contains a lateral channel, external memory.

Following Figure 5.6, denote T a component or channel data rate (throughput). T_{in} stands for the ingress interface throughput. It is obtained by summing up all input interfaces' data rates. T_{hw} stands for the worst case hardware acceleration part data rate[3] and T_{mem} is the maximal memory data rate. For the PC part T_{bus} is the PCI throughput and T_{swa} is the data rate for the software aggregation part.

[3]In this case it is assumed that the external memory throughput is infinite.

Abbr.	Description	Abbr.	Description
T_{in}	Input throughput	R_{in}	Input packet rate
T_{hwa}	Hardware aggregator throughput	R_{hwa}	Hardware aggregator rate
T_{mem}	Ondboard memory throughput	R_{mem}	Onboard memory context rate
T_{bus}	PCI bus throughput	R_{bus}	PCI bus flow rate
T_{swa}	Software aggregator throughput	R_{swa}	Software aggregator flow rate
S_P	Packet size		
S_C	Context size		
S_F	Flow record size		

Table 5.5: Throughput and rates for the analysis of a generic flow processing system

Similarly, a Protocol Data Unit (PDU) rate of a component or channel is denoted R. It then follows that $R_{PDU} = \frac{T}{S_{PDU}}$, where S_{PDU} is the PDU size. For interface data stream (T_{in}), S_p will denote size of packet; for the memory channel, S_c stands for size of context[4], S_f stands for size of a flow record. The defined symbols are arranged in Table 5.5. The aggregation factor can then be defined as the number of received packets divided by the number of packets produced by the aggregation unit (hardware or software), per time interval:

$$A = \frac{R_{in}}{R_{hw}}. \tag{5.1}$$

In order to process incoming traffic without packet loss, the hardware processing part must (a) provide at least minimal aggregation not to overwhelm the software aggregator and (b) must be able to handle incoming traffic, i.e. $T_{hwa} \geq T_{in}$. Because the hardware processor architecture is not known yet, let's assume that it is able to process T_{in} without packet loss and generates $R_{hw} = \frac{1}{A}R_{in}$ flow records per second. In order not to overload the software part it must hold that

$$R_{hw} = \frac{1}{A}R_{in} < R_{sw}, \tag{5.2}$$

where $R_{sw} = \min(R_{swa}, \frac{T_{bus}}{S_f})$. R_{sw} is the limit of software processing, which can be either the software aggregator or bus throughput. R_{swa} is given and it is the maximal number of flow records which may be processed by the software. Bus flow rate depends on its throughput and flow record size. Solving for A in equation 5.2 gives the minimal aggregation rate

$$A > \frac{R_{in}}{R_{sw}}.$$

Because the size of flow record may differ from the size of packet, the requirement can be rewritten as

$$A > \frac{T_{in}}{T_{sw}} \frac{S_F}{S_P}. \tag{5.3}$$

Similarly, hardware processing limitation, dependent on the memory throughput can be derived. If the memory must hold S_C bytes of context for each flow and its bidirectional throughput is T_{mem}, it can handle $R_{mem} = \frac{T_{mem}}{S_C}$ contexts per second. In the worst case, for each packet, one read and subsequently one write operation is triggered. In order to

[4]We will refer to the term *context* as part of a flow record that must be updated for each packet. Other flow record items, such as flow keys are static and thus do not change with each packet.

		Packet Size				
		84	100	300	700	800
Context Size	16	84	100	300	700	800
	32	42	50	150	350	400
	64	21	25	75	175	200
	128	10.5	12.5	37.5	87.5	100

Table 5.6: Maximal ingress throughput limited by memory (16 Gb/s).

handle an incoming packet rate (Figure 5.6) without loss, it must hold that $R_{mem} \geq R_{in}$ and therefore

$$T_{in} \leq T_{mem}\frac{S_P}{S_C}. \tag{5.4}$$

To sum up, Equation 5.3 sets the minimal aggregation factor for the hardware design, in order not to overwhelm the software part. Equation 5.4 sets the input throughput limit imposed by the onboard memory. Both cases assume that a hardware processing mechanism used is able to process incoming data without any packet loss.

5.6 Discussion

We will now summarize the information gathered in this chapter. Firstly, the onboard memory limitation will be derived from Section 5.5. The onboard memories considered on the COMBOv2 platform have throughput of 16 Gb/s (in both directions). According to Equation 5.4, the worst case condition limits the input throughput to

$$T_{in} \leq T_{mem}\frac{S_P}{S_C},$$

where S_P is size of packet at the ingress interface and S_C is size of contexts stored in memory. Table 5.6 shows limits for various context and packet sizes. Two possibilities are the most important ones: One with the context size of 32 bytes and the shortest packet size, defined by the IEEE 802.3 standard as 84 bytes[5]. This selection limits input throughput to 42 Gb/s, which is sufficient for the aim of this thesis. With doubling the context, the available channel capacity is 21 Gb/s, that is, the half. From this we can conclude, that the onboard memory throughput is not a limitation for up to 40 Gb/s flow monitoring.

Let us now consider the minimal aggregation factor defined in Section 5.3. We will now derive throughput limitation with respect to software layer capabilities and the results gathered from the simulations of a live traffic sample. The minimum aggregation that the hardware part must provide is

$$A > \frac{T_{in}}{T_{sw}}\frac{S_F}{S_P}.$$

For a 20 Gbit/s case, with 96 bytes flow record size, the filled equation gives

$$A > \frac{20}{7.68}\frac{96B}{S_P}. \tag{5.5}$$

The software processing limit has been estimated to 7.68 Gb/s.

[5]The standard defines the minimum frame size as 64 bytes, but 8 bytes of preamble and Start Frame Delimiter, together with 12 bytes inter-frame gap must be considered.

The exponential line in graph in Figure 5.7 shows the minimum aggregation required in order to process all incoming packets without loss. Also, both aggregation measurements are considered in the graph: the direct indexing algorithm and hash table. These are depicted as constant lines according to the simulation results from Section 5.2.

The direct indexing algorithm, with an aggregation factor of 3.7, preprocesses the incoming traffic sufficiently enough to be handled by the software part, starting from 67 bytes long packets. Hash table performs even better and therefore provides the same functionality. From the results we can conclude, that simply using direct hash indexing for 20 Gb/s networks, the two-layered hardware-software solution can measure network without packet loss even on shortest packets[6].

One should also note the area in Figure below a minimal aggregation of one. This value defines the region, where a software solution without any hardware preaggregation is able to process data without packet loss. In this case, it is estimated, that software measurement probe would be able to process 20 Gb/s traffic without packet loss if an average packet rate would be more than 250 bytes. The hardware preprocessing would just require stripping the date off payload and sending packet headers into software for further processing.

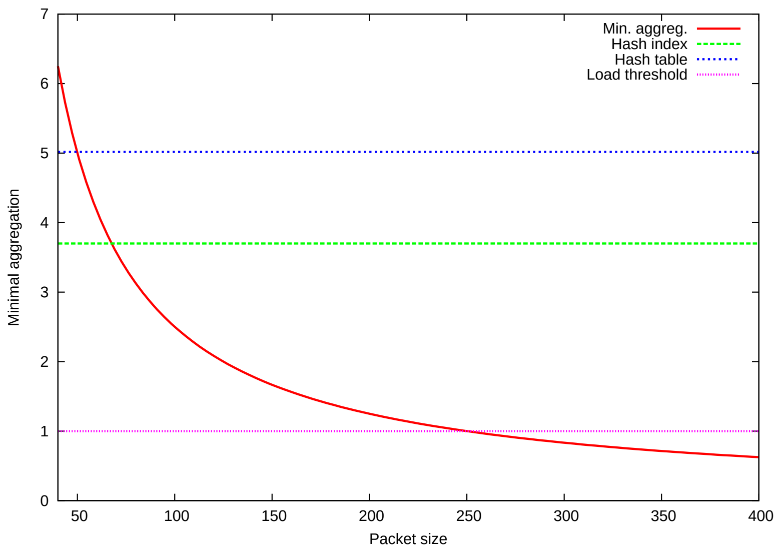


Figure 5.7: Minimum aggregation for a 20 Gb/s fully loaded network, depending on average packet size. The exponential line shows minimal aggregation according to formula $\frac{20Gb/s}{7.68Gb/s}\frac{96B}{S_P}$. Size of flow has been set to 96 bytes, software limit has been set to 7.68 Gb/s (10 Mpackets/s $\times$ (96 $\times$ 8) bits per packet).

[6]Shortest packets are considered as 84 bytes, according to IEEE 802.3 standard. 64 Bytes is the frame contents, 8 bytes preamble and start frame delimiter, 12 bytes is the inter-frame gap.

Chapter 6

Architecture

This chapter introduces the hardware accelerated architecture for flow monitoring. The analysis in the previous chapter showed limitations mainly of software probes, which introduce significant packet loss when dealing with rates at or beyond 10 Gb/s. Therefore hardware solution is necessary in order to accelerate this high speed monitoring. The following points shortly summarize the results from Chapter 5:

- The theoretical limits of a current pure software solution are approximately at 10 million packets per second. Such solution barely manages to monitor a fully loaded 10 Gb/s link. If monitoring in both directions of such configuration would be necessary, it would not suffice and serious packet loss would occur. However one advantage of SW solutions is that a very large memory can be allocated in order to hold as much flow records as possible in order to achieve reasonable aggregation factor, defined by a traffic mix.

- Hardware platforms have very limited resources in terms of flow cache size, but on the other hand the speed of processing is very high. Thus, very small aggregation can be achieved when compared to software solutions.

- The requirements on the flow aggregation structure are very diverse. Not only due to IPFIX definitions, but also because every application that performs as a collecting device might require its own flow record definition. Therefore the monitoring application must not only be hardware accelerated, but also meet different user demands.

It is therefore necessary to combine both solutions in a system that is able to preprocess data at high rates, but on the other hand, provides a reasonably large flow storage to achieve maximal aggregation and with maximum flexibility. Figure 6.1 shows the basic concept. Incoming data is handled by the hardware part which preaggregates it sufficiently enough for the software aggregator to be able to handle the incoming traffic flow. While both hardware and software parts must cooperate tightly, the task of this thesis is only design of the hardware accelerator part.

The system-level architecture proposed here is a high-level pipelined system. Each stage processes its incoming chunk of data and passes the result to the next one. The architecture is based on the NetCOPE platform (Section 5.4.1) that provides and abstraction layer both from the side of ingress interface and software layer. Note that because the probe is passive, there is no egress interface.

The previous chapter identified that when dealing with high packet rates, both the PCI bus and processor become a bottleneck in flow monitoring. Because the most critical part

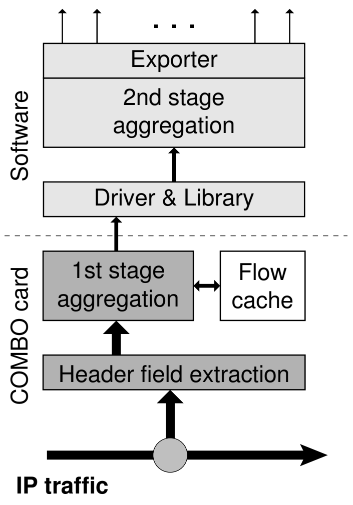


Figure 6.1: Main concept of hardware accelerated flow monitoring targeted at 10 to 40 Gb/s.

is the flow update, with almost 60 % system load under a worst condition, the architecture was chosen to include also the aggregation process[1].

When a packet enters the system core (the part stripped off the NetCOPE layer), it is first preprocessed by a *Header Field Extraction* block. The packet's header fields are analyzed and proper information extracted into an unified data structure which is passed on to the next processing stages. This unified structure (Unified Header, UH) contains flow keys and data necessary for flow aggregation. A hash identifier is computed from the key fields in the next block. It is then appended to the UH and passed on. The result of this preprocessing stage is data ready for aggregation, in its most compressed form. The most important part is the one that aggregates data. It must decide which flow to pick up from a memory, or which flow to reject when the memory is full, and provide the aggregation process. The last part in the system is the inactive management part, which ensures that short-lived flows are exported as soon as possible.

To meet the design criteria almost every part of the system is configurable. However, the system is not a general purpose FPGA based processor, because such structure would be too slow and consume much more resources in the chip. The system characteristics are defined at compile time and a fixed FPGA configuration is then created. In case there is a need to quickly change the functionality of the system, several variants must be precompiled.

The system design is also targeted for code reusage. As much as it is possible, IP cores are used for the architecture implementation and their design is also taken into account. This chapter also presents some improvements of the IP cores used in order to boost processing rate.

6.1 Data structures

The components described in previous section exchange data between themselves. In this section, we will describe the structure of this data, as it is necessary for further architecture

[1]In real traffic without anomalies, this might not be necessary, because as the analysis shows, real 10 - 40 Gb/s traffic contains very low packet rates when compared to the link protocol maximum.

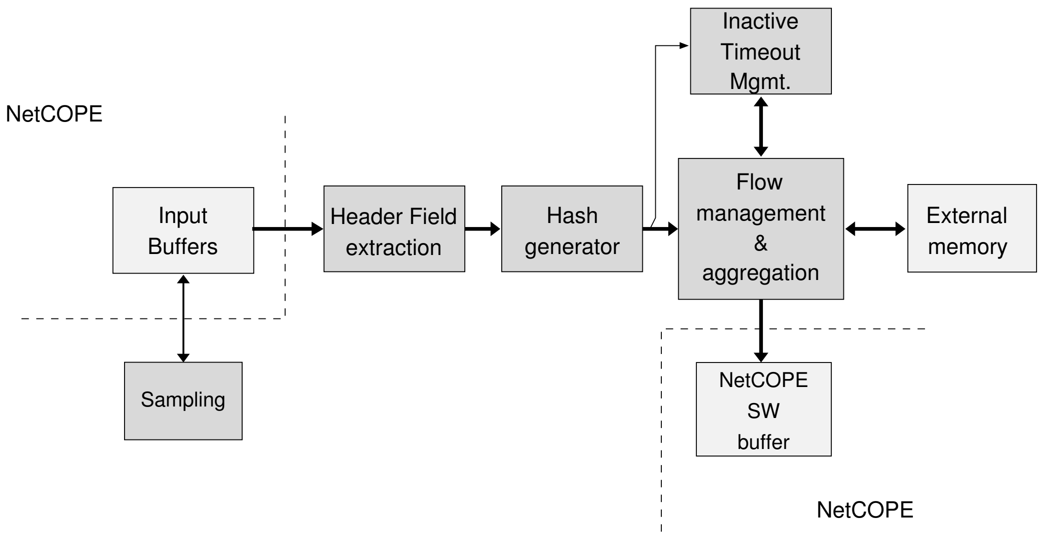


Figure 6.2: Firmware of the proposed flow monitoring probe.

description and to define the software interface.

Three fundamental structures are defined (Figure 6.3):

- *Unified header* carries protocol header fields that are necessary for flow processing. It contains fields extracted from the packets' protocol headers. The structure is passed throughout the system until processed in the flow processing unit.
- *Flow record* fully describes one flow unit, where statistical data are aggregated. It contains (i) flow key fields and (ii) aggregated statistical data. Figure 6.3b shows an example of a flow record for the Netflow version 5 protocol. The shaded fields are flow identifiers, which do not change during the flow lifetime. These are therefore immediately exported to software because are not necessary to be stored in an onboard memory.
- *Flow context* is a portion of flow record stored in onboard memory, together with a 64 bit hash identifier. In fact, it is a flow record in which flow key fields are replaced with a hash. In the Figure, context is formed by remaining, unshaded elements.

According to this definition, software layer therefore receives two kinds of data: a flow key identifiers and after a flow has been exported from the onboard memory, it receives the flow context as well. This way the onboard memory capacity can be fully utilized to hold as much flows as possible.

6.2 Packet capturing and preprocessing

6.2.1 Packet data preprocessing

The task of header field extraction block is to parse the incoming packet flow, decode the protocol structure present in the packet header fields and extract appropriate information from the data flow. The extracted data is then packed to an unified structure that is processed by subsequent blocks.

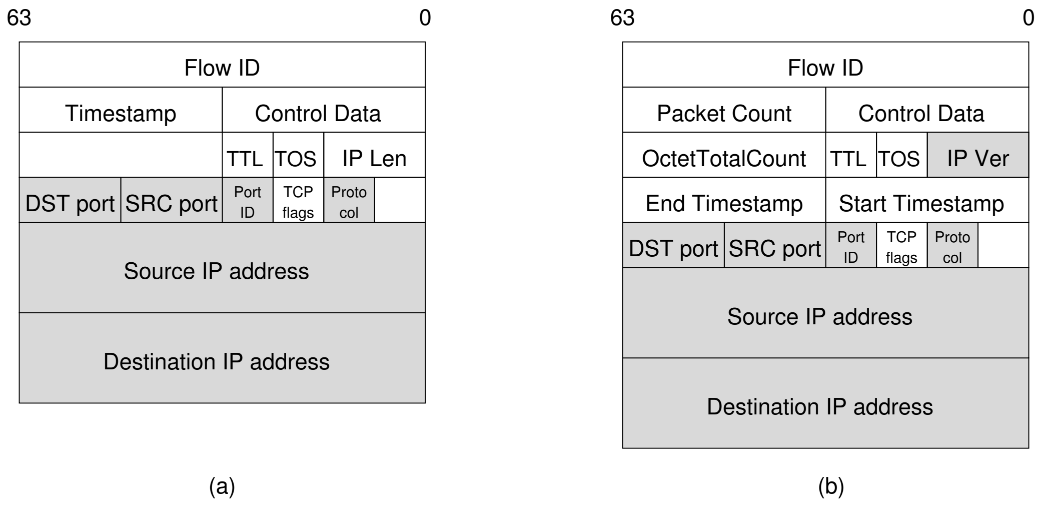


Figure 6.3: Unified header structure and flow record/context structure.

There are several options how to choose the best processor. Firstly, a general purpose processor implementation can be created or an IP core used. Possible candidates could be a Xilinx Microblaze or Picoblaze processor [48] or a RISC Header Field Extractor processor (HFE) developed by the Liberouter project [23]. Although such used components are on-the-fly programmable, they posses several disadvantages. The Picoblaze processor is 8-bit wide and with 100 MIPS it is not possible to achieve high throughput. The 32-bit wide icroblaze on the other hand contains a 32x32 bit general purpose registers, a fixed point multiplier and divider and FPU. Therefore such approach is a waste of resources which will never be used on the chip. The HFE processor is a specially designed 16-bit processor for hardware stream applications, however its throughput is insufficient for high speed processing (see Table 6.1).

In order to speed-up processing, an application specific processing engines must be used. Two solutions will be described here: *HandelC-based Header field extractor (HFE-C)* and *XML-based header field extractor (HFE-X)*.

HFE-C is a processor written in handelC and specially designed for network applications. It follows a macro-based approach, where each protocol is defined by a handelC macro. The processor is fully configurable via a handelC include configuration file. The throughput of the processor is much higher than a general purpose RISC processor already considered. An interesting comparison between HFE-C, RISC HFE and Microblaze can be found in [9].

HFE-X is another generic extraction engine developed at the Liberouter project [30]. It is highly configurable and can process data at nearly 10 Gb/s per extraction component. Therefore it might serve as one of the first candidates in the proposed monitoring system.

Table 6.1 summarizes main characteristics of extraction engines that were considered for this task. It is clear that general processors such as Microblaze or HFE have insufficient properties for monitoring at 10 Gb/s. Even the HFE-X cannot handle fully loaded 10 Gb/s link[2] and so all functionality must be parallelized in order to achieve reasonable throughput. One can read from the table that the most cost-effective solution could be a conjunction of two 32-bit HFE-X, running on 156.25 MHz clock.

[2]It must be noted that the results are for one specific configuration of components. The result for other configurations are not presented here for brevity.

Processor	Data width [b]	Frequency [MHz]	Throughput [Mb/s]	LUTs	BRAMs	Mbps/LUT
Microblaze	32	200.00	83	N/A	N/A	N/A
HFE	16	100.00	782	N/A	N/A	N/A
HFE-C	16	156.25	2,400	1600	1	1.5
HFE-X	32	156.25	5,000	1276	1	3.9
HFE-X	64	100.00	6,400	2665	3	2.4

Table 6.1: Available extraction engines. The resources are considered for Xilinx Virtex 5 architecture.

6.2.2 Hashing unit

The next processing stage is the hash generation. It receives unified header created by the extraction engine and generates a 64b hash. It then serves as an identifier and for indexing purpose in the flow lookup process. The hash is computed from the key fields of the flow (these must be present in the unified header). For the hash computation to be maximally resource-saving, no key fields extraction is performed. Rather the non-key portion of incoming unified header is masked out, so these fields do not influence the resulting hash value. The block structure of the component, which is fairly simple, is in Figure 6.4.

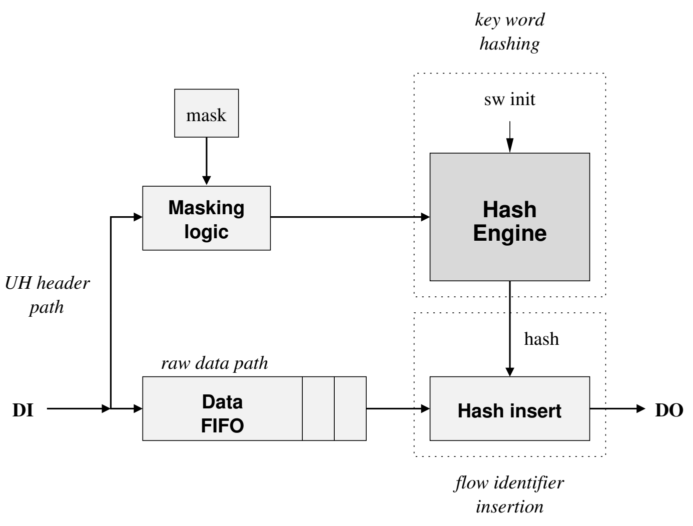


Figure 6.4: Hash generation firmware block.

Because the key fields selection is user configurable and is defined in the flow record structure, this hash generation component must reflect this. This is accomplished by a special *mask vector* preloaded or precompiled into the component.

6.3 Flow lookup

The most important and hardest task of the system is how to pick up a correct flow for a packet and update the flow record. The lookup scheme is defined by indexing algorithms, which were dealt with in Chapter 4. Two fundamental approaches can be used:

- The first one is a direct flow addressing based on the computed hash function. For each packet a hash is computed and part of the value used as an address into the flow memory. The advantage of such scheme is that it is very simple to implement and does not consume much resources on the chip. On the other hand, direct hash addressing suffers from a lot of collisions and therefore does not utilize the flow record memory fully.

- The other, more effective approach is to use a hash table. A portion of the computed hash serves as an address into the table. A bucket picked up by the address contains a list of flow records, which are then sequentially traversed in order to select the right item. The advantage is that the flow record memory is fully utilized, however the system throughput might suffer, because for each packet, several flow records must be read from the memory.

The simulations show that for a typical 10 Gb/s network[3] an approximate aggregation of 1:5 can be achieved with direct indexing (for more information please confer Chapter 5). This value defines the worst case, i.e. packets carry minimum amount of data. In such a case, software load would be approximately 4 Gb/s. The flow record rate would therefore be at most 6 million flows per second.

According to the result obtained by testing a current software solutions and taking into account the preprocessed nature of flow records entering software part, we can conclude that the proposed double-layered joint hardware-software system will in the worst case conditions be able to process full-duplex 10 Gb/s traffic without packet loss. However, care must be taken when assessing the capabilities of this system. As discussed in Section 5.5, flow context size[4], full flow record size and hardware limitations must be taken into account, because the flow record structure is user defined.

The indexing and flow distribution part is based on FlowContext [18], also developed by the Liberouter project. The block structure is in Figure 6.5. FlowContext is a generic system for stateful packet processing and therefore perfectly fits the flow monitoring task. The input to the system are analyzed packet headers in a form of unified headers. Part of it is the unique flow identifier, which is used as an address to the context memory. There are two main components of this system, we will describe their functionality briefly:

- *Context Manager* maintains memory integrity in the system. Because the processing path might have long latency, therefore two copies of a context may be present in the system: one copy in the external memory, the other in the processing unit. Context Manager must recognize the most up-to-date position of a context and issue or not a proper external memory request.

  The other task of this part is to balance the load within several processing units (PUs), which will be described later on. The load balancing is accomplished by fast on-chip associative memories.

- *Endpoint unit* provides an interface to the processing part. It is by means of a random access memory.

The interfaces of FlowContext are set to handle both incoming packet headers and the packet payload. However, in this thesis it is assumed that no payload processing is done.

[3]This gives 20 Gb/s in case of both directions monitoring.

[4]The portion of flow record size stored in on-chip memory. This is usually the record stripped of flow key fields. Flow key fields are replaced by a 64 bit flow ID to avoid collisions.

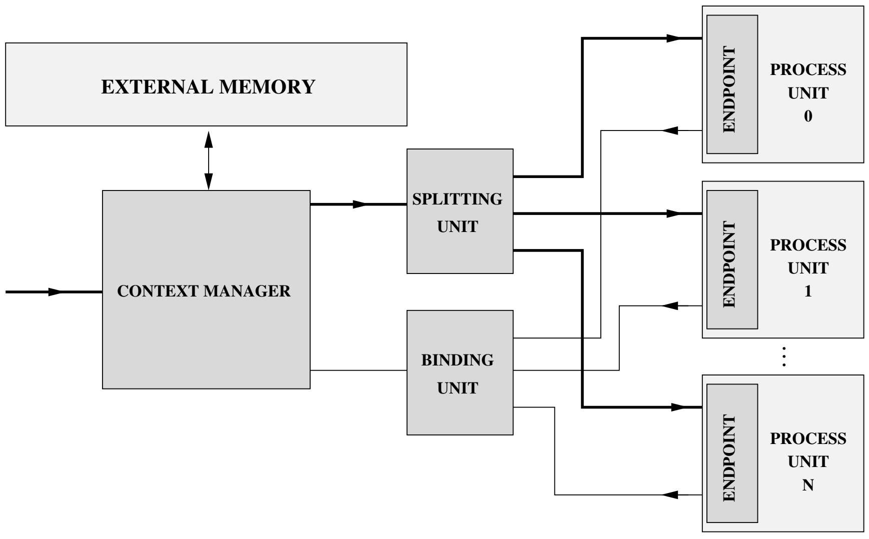


Figure 6.5: Flow context architecture. Figure courtesy of Martin Košek, [18]

Let's now have a closer look at the endpoint component. Its structure is in Figure 6.6. Here, the context, headers and payload memories have random access. From the Figure at can be seen, that the most loaded part is the context memory. In a worst case, it must (i) receive a context, (ii) provide the read and write interface for the processing unit and (ii) send context to a context manager to maintain context integrity. Thus, the context memory requires two read and two write ports, all independent of each other. However, building such memory might not be possible because it might use a lot more resources than necessary. If only a two port RAM is used for context memory, it is not possible to utilize the bus fully and so the performance would suffer.

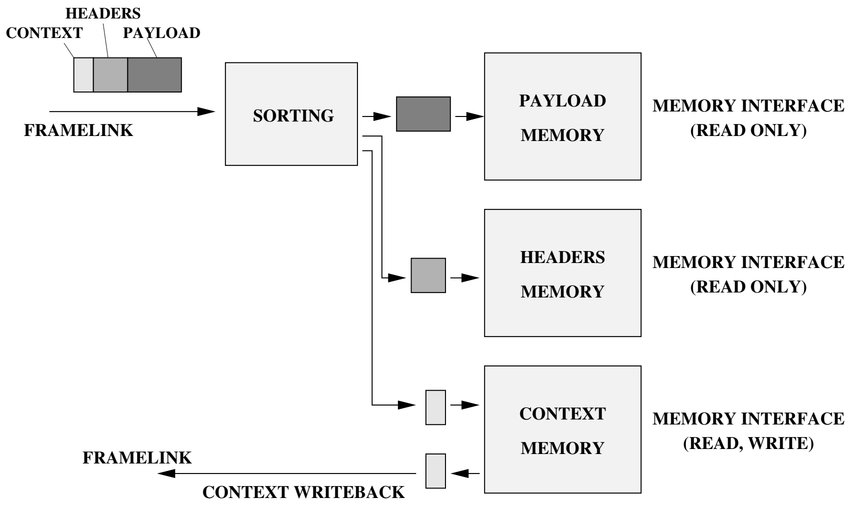


Figure 6.6: FlowContext endpoint block structure. Figure courtesy of Martin Košek.

This model, however, can be optimized, under some assumptions. Firstly, we will assume that the processing part (see Section 6.4) is designed to fully pipeline an incoming flow of headers and contexts. The other assumption is that the processing latency is lower than the context write latency from Figure 6.6. Then we can derive an endpoint model in Figure 6.7. Note that there are no header and context memories. If the processing unit is not a bottleneck of the system, these are not necessary. Another thing is that the context memory has been moved just *after* the processing pipeline and form a feedback loop to the pipeline input. This is necessary in order to handle flow bursts. Also the writeback part is no longer connected to the context memory, but the writeback data is in parallel transferred into the binding unit (Figure 6.5).

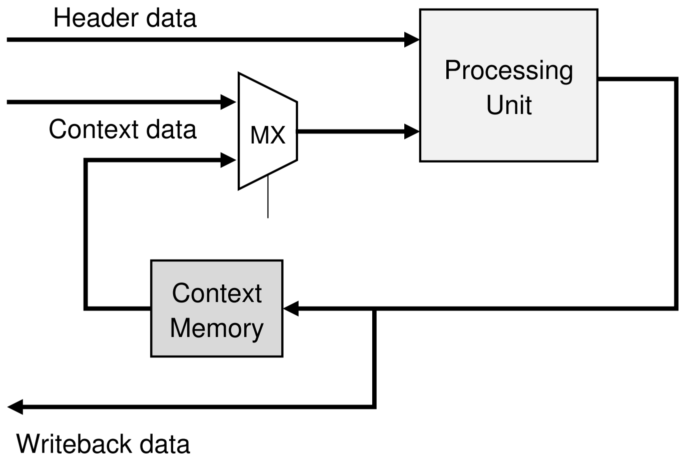


Figure 6.7: Optimized endpoint model, to fully utilize context and header buses.

6.4 Flow update

The flow update task is handled by a flow *Processing Unit (PU)* a specially designed arithmetic and logic unit (ALU) to meet the flow update process characteristics: high throughput and stream processing. Because we want the system to be fully user-configurable, a PU must also be able to handle user flow record definition.

The header and context structure used is as defined in Section 6.1: context is a data structure stripped off flow fields, with a unique hash stored with each record to detect memory collisions. We can describe the functionality of a PU in the flow chart in Figure 6.8. The processing starts with reading both context and header (and optionally packet payload) and checking for collisions. In case of collision the old context is exported to software, together with flow keys of a new context. The keys are not stored in the memory as discussed previously, but instead a hash is stored. Another case is if the packet is the context's first, then the default values are provided. Subsequently, the actual update operations are executed and the updated context checked for overflow. Accordingly, the overflow flag is stored in the context and it is saved into the memory of updated contexts and also a writeback request sent to the context manager.

As the processing core, and automated generator of processing elements for FPGA [22] has been chosen. It provides the functionality needed for this task, is highly configurable and targeted at high-rate applications. The selected component is a general purpose processor, that is unaware of our flow processing task, although care has been taken to design it for this monitoring probe. In order to tune its performance, these amendments are proposed:

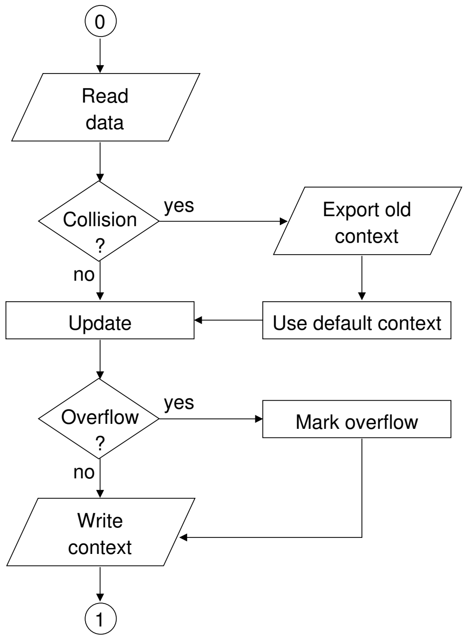


Figure 6.8: Flow Processing Unit flow chart.

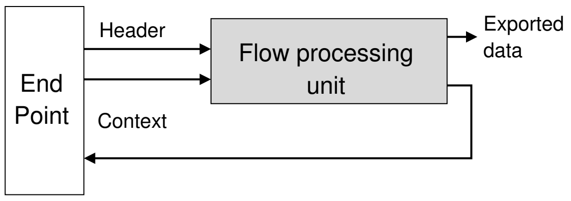


Figure 6.9: Flow Processing Unit interface.

- In order to reduce the amount of context stored in an onboard memory, the component must be aware of different flow record and flow context structures.
- After an update, when an overflow is detected, the context is not immediately exported to software, but is rather stored back with a *forced collision* flag. When the context is read out once again, it forces a collision and its export. This action simplifies expiration policy and saves chip resources. It does not harm performance in any way as well.

Both amendments are derived from the optimized FlowContext endpoint structure (Section 6.7) where a flow burstiness must be taken into account.

6.5 Inactive timeout management

The purpose of inactive timeout in this context is to identify flows which have not been updated for a specified amount of time. Inactive flow records are then transferred to the software part. From the implementation point of view, keeping an activity state for records might be accomplished in several ways. However, the proposed algorithm for this task in the view that the system contains only direct indexing, tries to be as cost effective as possible.

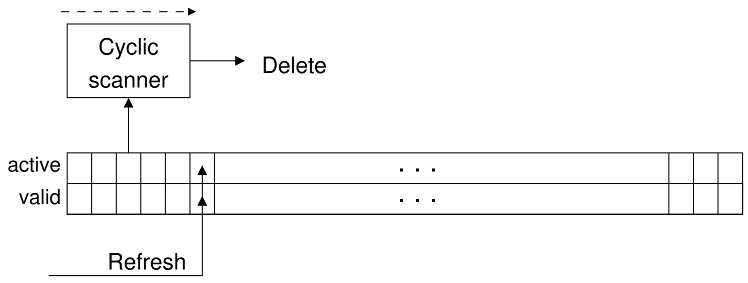


Figure 6.10: Inactive timeout management.

The algorithm uses an on-chip memory to keep the activity vector in (Figure 6.10). For each record in the flow cache, two bits are stored in the memory: a valid bit and an activity bit. There is also an one-to-one mapping between the address spaces of activity memory and flow cache. The component consists of two parts:

- *Activity refresher*, that sets both the valid and activity bits for each incoming packet.
- *Inactivity cyclic scanner* traverses periodically the activity memory and checks if a record has its activity bit set. If yes, it unsets it; if not it schedules the record for an export to the software and clears the validity bit.

The system starts with initializing its memory with all flow records invalid which correctly corresponds to an empty flow cache; then the traversal starts.

This algorithm is fairly simple, but introduces a significant error when large timeout values are used. In fact, with timeout set to T seconds with traversal period $\frac{T}{2}$, the real inactive timeout value will be in range $[\frac{T}{2}, T]$. The worst case situation leads to an activity bit being set by an incoming packet, and then immediately reset by the scanner, which will then free the item in next round. Thus, the item would be exported after $\frac{T}{2}$ seconds of inactivity instead of T. In our case, however, such error is acceptable.

6.6 Flexible flow record definition

6.6.1 Record definition files

One of the design requirements of the probe is to handle the diverse configuration needs. Therefore, as already stated, almost every components is user-configurable. This configuration is done during compile time.

A user defines his own monitoring process by defining the specific configuration of the system's components. To provide a user interface a XML definition schema has been created. This idea is not new and has already been proposed in prior work [45]. Because the definition itself is not the main aim of this thesis, the XML structure will be only briefly defined here. Full definition of the configuration file can be found in [45].

The configuration file contains all necessary information for the system to be able to derive configuration for all its components. Its structure defines (i) Unified Header, (ii) flow record and (iii) context update operations (Figure 6.11).

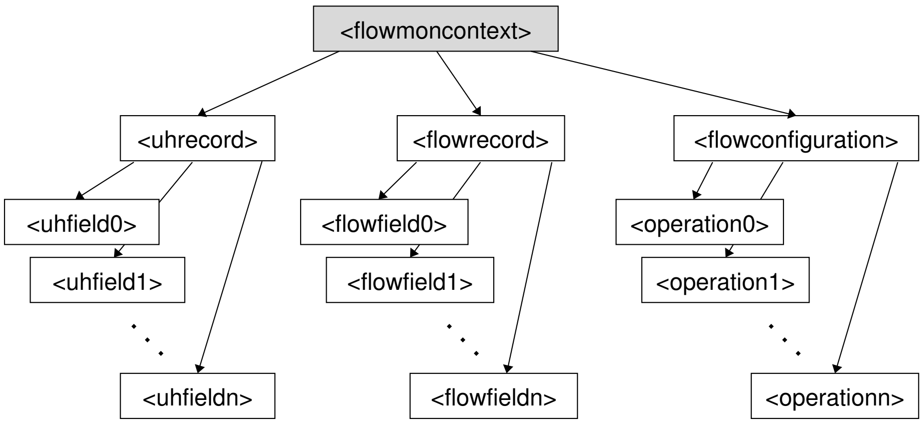


Figure 6.11: The probe configuration structure.

6.6.2 Design generator

The part of the probe configuration process is a software generator. Its aim is to parse input XML files and distribute the monitoring process definition among configurable components.

The structure of the generation engine is in Figure 6.12. The program reads and parses XML description files, checks them for errors and generates these outputs:

- For the HFE component a configuration include file is generated. Then the handelC compiler is run and an implementation created, which will be used in the synthesis process.
- A mask for the Hash Generator component is created in the component properly configured to reflect flow key fields that enter the hashing process.
- Implementation files for Flow Processing Unit.

The generated codes are then used in a synthesis process which outputs bitstream configuration loadable into FPGA.

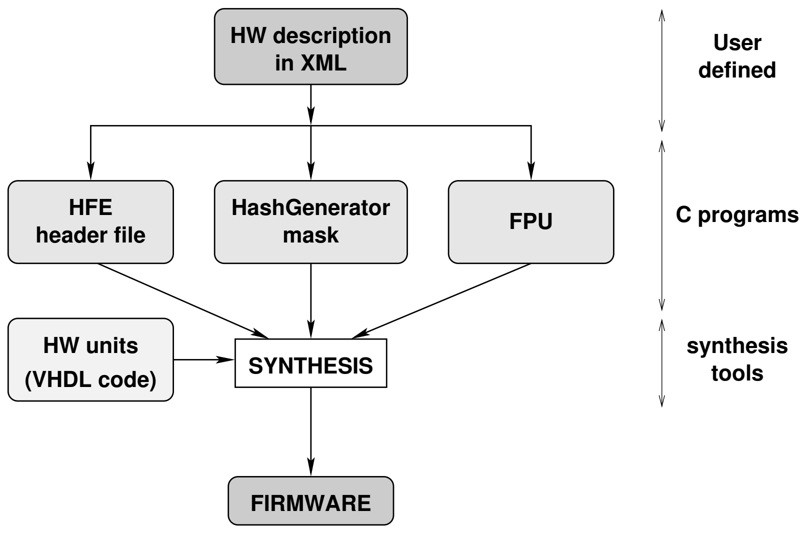


Figure 6.12: Firmware generator program structure.

Chapter 7

Implementation and results

This chapter presents an implementation of the designed architecture. Because the actual prototype platform is not the COMBO version 2 family of cards, the specific hardware features are firstly described. Then a brief description of the system is given.

The implemented functionality of the design has been thoroughly tested at the local university network so the whole system is prepared for long lasting sustained measurement. In the final section, the throughput of the system is presented.

7.1 Implementation

7.1.1 Hardware platform

In this Section a prototype architecture will be presented. All components described so far have been implemented in VHDL langauage, suitable for hardware description.

It has been defined that the target platform be a COMBO version 2 family of cards. However, during the implementation phase, the hardware was not yet available. In order to build and run the prototype, COMBO6X family was chosen, which is a previous development version.

The COMBO6X family is similar to the one described in Section 5.4. It consists of a mother and interface card, the mother card contains the core of the system, while the latter one provides a specific network interface connection. Both cards contain a Xilinx Virtex-II Pro FPGA, which can be utilized for processing. At last, the cards are connected via a 64 bit data interface, running at 100 MHz. The mother card also contains three SSRAM memories with 32 bit/100 MHz interface.

Because the card is equipped with NetCOPE platform, which handles low level ingress network processing (packet and CRC check, etc.) and data transmission to software, the implementation is very much hardware-independent. The created implementation can therefore be very easily transferred to any hardware supporting the NetCOPE platform.

7.1.2 Monitoring part

Figure 7.1 depicts implemented architecture. The processing is divided among both cards. The extraction part takes place at the interface card and produces Unified Header. These are then transported via the 64b inter-card connection into the mother card, where the rest of the processing pipeline is located. For the implementation, the HFE-C processor has been selected, because it still offers satisfactory throughput at reasonable cost. Eight HFE-C units were used for the extraction task.

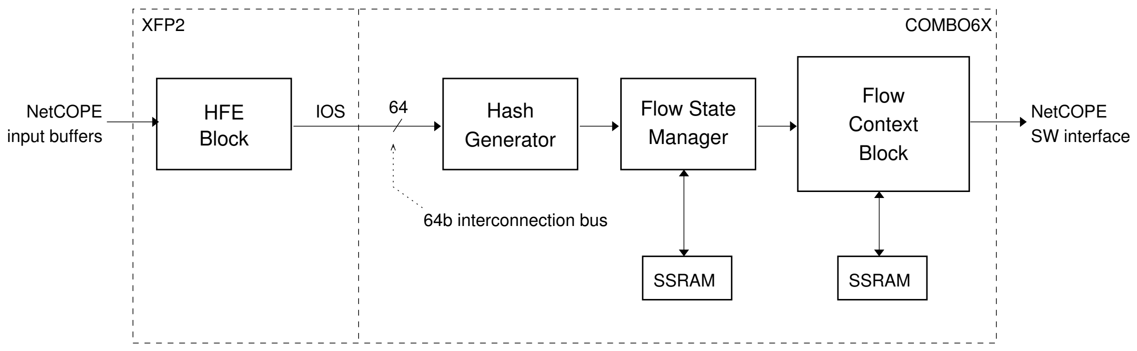


Figure 7.1: System prototype on the COMBO6X and XFP2 card.

As the flow cache, two SRAM memories are used in this configuration. The memories are used in parallel and offer a total of 6.4 Gb/s throughput in one direction. Because the read and write operations must share the same bus[1], the resulting throughput must be divided by two, resulting in 3.2 Gb/s.

The third SRAM memory is occupied by the inactive timeout manager, which must keep state of each flow.

The flow update part (flow context block) has two processing units instantiated. because the memory limits the maximal throughput of the system, the selected number is sufficient for this configuration. Since the architecture has been implemented as a scalable system, the number of processing elements can be set by user, according to his specific needs.

7.1.3 Functionality

This thesis deals with the hardware part of a whole monitoring system, comprising the hardware and software aggregation layer and the collecting device. The main aim is to divide the processing load between these two layers in order to be able to handle high packet-rate traffic. Because the second aggregation layer is not ready to be used yet, the probe functionality has been tested without this layer. Although such setup without secondary aggregator certainly produces a lot of collisions because of a small capacity of the flow cache, it still provides sufficient aggregation level.

The probe has been monitoring sustained 10 Gb/s interface at the Masaryk's university local network for two weeks without intervention. The probe has been set up according to Figure 7.2. A tapped traffic enters the hardware accelerated monitoring part where primary aggregation takes place. The data is then transferred to the host PC and is immediately exported to a remote collector. In this setup, a nfdump collecting device has been used to store the data and a nfsen web interface for visualization. NFDUMP is a set of free network flow processing tools [15].

The results of the probe were compared with a stable version of another measurement device which output is considered reliable. This test scenario showed that

- the implemented device is stable and is ready for a long-term deployment, measuring

[1]It is the limitation imposed by the memory.

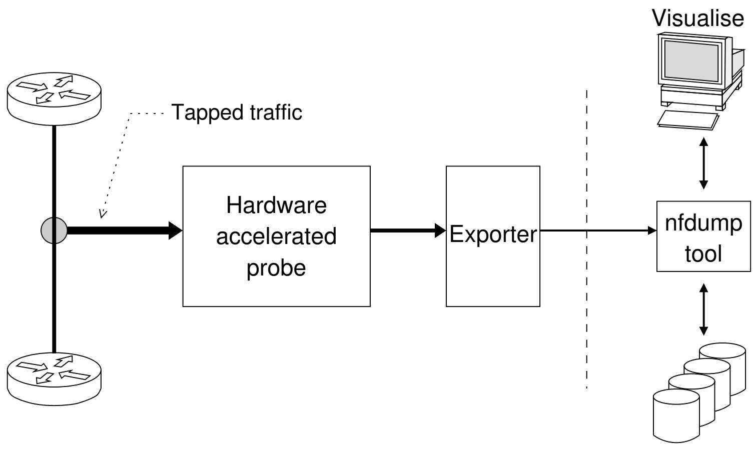


Figure 7.2: Probe functionality test setup.

with Netflow version 5 flow record definition (although any other record structure may be configured);

- the probe is capable of a high-rate processing, limited only by the underlying platform used. The tested throughput limitations are discussed in Section 7.2.

7.2 Throughput

In order to derive performance of the implemented monitoring probe a measurement has been carried out using a generator of synthetic network traffic. The generator is capable of sending a user defined packet flow with rates of up to 10 Gb/s per interface.

The probe has been set up to save 64 bytes of context for each flow and the flow record size was also 64 bytes. The onboard memory can therefore hold up to 128,000 simultaneous flows, although the real number will be less than that because of collisions. For the context update part, two Flow Processing Units were used.

The measurement comprises sending a specified amount of flows to be processed which is the parameter of the test. For each flow count, the throughput is measured as the number of packets sent, divided by number of packets processed. The traffic generator has been set up to generate packets with shortest possible length (64 bytes, comprising Ethernet frame with CRC checksum) with highest available rate. Thus, the worst case has been measured. The packets lost at ingress interface of the probe are those discarded due to probe overload, or error packets. Because no error packets were observed during the test, the measurement may be considered accurate.

The graph in Figure 7.3 shows throughput of the probe depending on number of simultaneous flows generated. The graph contains two distinct regions: the first one shows a mild peak with flow count less than approximately 100 flows. The peak throughput is around 4.2 Gb/s which is currently a limitation of the two FPUs used. This behavior, which suggests that for small number of flows, the throughput is slightly bigger than for the rest of the graph. This is probably caused due to the fact that the FlowContext system contains a small high-throughput memory located in Endpoint Components. This cache memory therefore bypasses SRAM reads and writes and provides higher throughput, which in this case is limited only by the throughput of FPU components. From the results it can be derived that the raw throughput of one FPU is around 2.1 Gb/s in this configuration.

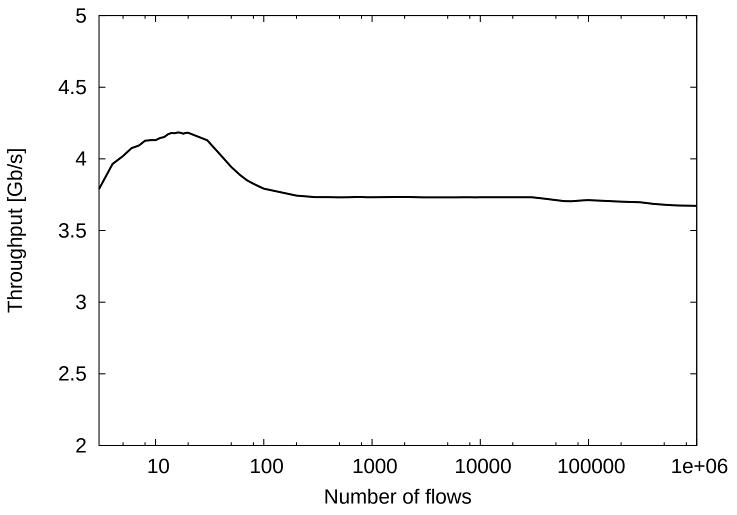


Figure 7.3: Throughput dependent on number of flows sent. Synthetic traffic generator has been used. Size of context was 64 bytes. Shortest packets were sent at full 10 Gb/s rate.

When reaching 100 simultaneous flows and more, the system's throughput stabilizes at around 3.7 Gb/s, which can be considered the limit of the SRAM onboard memory. Recall that although the SRAM throughput has been estimated to 3.2 Gb/s an actual measured value is higher because for each packet, and inter-frame gap and preamble must be accounted for, which utilizes the link only at 76 % at shortest packets.

Although the test shows that the probe achieves higher throughput for small number of flows, such measurement is probably irrelevant for live-traffic measurement, with several tens of thousands of simultaneous flows per second. Therefore the region at the right hand side of the graph must be considered as an accurate and usable measurement result.

One other thing should be noted when regarding the variability of the context size. The measurement in this case is strongly dependent on the context size stored in memory, because the time needed to read and write a context is crucial for the throughput of the probe. Thus, with a 32 bytes long context, the throughput could be effectively doubled.

Chapter 8

Conclusion

This thesis dealt with a design and implementation of a high-speed network monitoring probe. The probe performs a so called flow-based monitoring, a measurement technique based on a notion of communication between two end-nodes in a network, and gathers statistical information about packets passing a measurement point and classified to belong to certain flow. The main aims of the thesis, i.e. analysis of principles behind flow monitoring, design of the probe, and its implementation, have been accomplished. The result of the work is a fully functional device. Moreover, the device has been tested at a local university backbone network, with peak rates of up to 5 Gb/s. The testing shows that the probe is capable of providing a long-term measurement without packet loss on a 10 Gb/s network.

Most flow-based monitoring probes are based on Netflow protocol, supporting its several versions. Whereas Netflow version 5 defines a static flow record, Netflow version 9 is a step forward in that it allows a user defined flow record specification and is thus more flexible. Since it is very important to follow widely used standards, the approach in this thesis takes IPFIX as its fundamental information elements fields definition. IPFIX is an emerging standard based upon Netflow version 9, but it is open and allows for user proprietary extensions and thus is more flexible than Netflow.

Because a monitoring process might not necessarily be based on a classical five-tuple description (source and destination IP address, source and destination port, protocol), one of the main aims was to design the probe as a *flexible* device. A user is allowed to define his own monitoring process via a sophisticated XML-based flow record description. The specification can be then used for an automatic generation process of the hardware accelerator. The whole process is automatic and requires only that the user provides the description.

Another important aspect of the design is capability of high-speed processing. Pure software implementations are not able to perform monitoring without packet loss at high packet rates. Moreover, with rates of more than 10 Gb/s the system bus becomes bottleneck as well. Thus, this thesis also focuses on a hardware accelerated architecture. The hardware part performs first stage aggregation and produces intermediate flow records, which are then further aggregated into complete flow information ready to be analyzed or stored. The hardware part is fairly simple and cannot provide full aggregation. However the system does not suffer from packet losses and the software part can still provide nearly ideal post-aggregation in the second stage.

The resulting system thus combines support for three widespread export protocols: Netflow version 5 and 9 and IPFIX, allows high flexibility through the possibility of a user defined flow record structure and provides monitoring of high-speed networks, starting at

10 Gb/s.

The results obtained from extensive probe testing show that the probe is capable of processing up to 5.5 million packets per second without any loss. Although the performance does not reach 10 Gb/s at shortest packets, the limitation is not in the system's design, but in the hardware platform used. If using the target, COMBOv2 platform, the probe will be able to process up to 40 Gb/s traffic without packet loss, provided a 32 bits long flow context would be selected.

The applicability of the probe in networking is diverse. One of the simplest applications is operational monitoring and usage based billing. Usage based billing may require the probe be capable of high rate data processing to provide accurate pricing for customers. Sampling might in this case require difficult reconstruction algorithms to ensure the pricing data is accurate.

Recently, network traffic classification becomes one of the primary interests for administrators as well as academic community. It as been mainly due to the fact, that many applications use dynamic port allocation and therefore the basic port-based classification techniques are no longer applicable. However, traditional intrusion detection systems and classificators might not work as well, as network traffic is becoming increasingly encrypted and thus only the IP and transport layer information is visible. Flow-based monitoring in this case provides a useful source of information because it is capable of incorporating almost any data or timing information mined out from data representing particular flow. Common applications include traffic classification, with Voice over IP and similar being recently especially interesting. Some other examples may be anomalies detection, intrusion detection and DoS-based attacks. Monitoring of all these activities has one property in common – a need for high-speed data processing and flexibility of the monitoring device. The solution presented in this thesis offers both, thus possibly easing work of many developers and researchers.

Bibliography

[1] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for Traffic Engineering Over MPLS. RFC 2702 (Informational), September 1999.

[2] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFCs 1349, 4379.

[3] S. Bradner and A. Mankin. The Recommendation for the IP Next Generation Protocol. RFC 1752 (Proposed Standard), January 1995.

[4] CAIDA. The Cooperative Association for Internet Data Analysis. `http://www.caida.org`.

[5] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.

[6] Cisco. Cisco NetFlow. `http://www.cisco.com`.

[7] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[8] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), January 2008.

[9] T. Dedek, T. Martinek, and T. Marek. High level abstraction language as an alternative to embedded processors for internet packet processing in FPGA. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 648–651, Aug. 2007.

[10] S.E. Deering. Host extensions for IP multicasting. RFC 1112 (Standard), August 1989. Updated by RFC 2236.

[11] L. Deri. nprobe: an open source Netflow probe for gigabit networks. In *Proc. of Terena TNC*, 2003.

[12] N. Duffield. Sampling for passive internet measurement: A review. *Statistical Science*, 19:472–498, 2004.

[13] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[14] Y. Gu, A. McCallum, and D. Towsley. Detecting anomalies in network traffic using maximum entropy estimation. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[15] P. Haag. NFDUMP – Netflow Processing Tools. Available online at `http://sourceforge.net/projects/nfdump`.

[16] S. Halevi and H. Krawczyk. MMH: Software message authentication in the Gbit/Second rates. In *FSE '97: Proceedings of the 4th International Workshop on Fast Software Encryption*, pages 172–189, London, UK, 1997. Springer-Verlag.

[17] INVEA-TECH. INVEA-TECH FlowMon. `http://www.invea-tech.com`.

[18] M. Košek and J. Kořenek. Flowcontext: Flexible platform for multigigabit stateful packet processing. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 804–807, Aug. 2007.

[19] S. Kundu, S. Pal, K. Basu, and S. Das. Fast classification and estimation of internet traffic flows. In *Passive and Active Network Measurement*, pages 155–164. 2007.

[20] J. F. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[21] K. Lan and J. Heidemann. A measurement study of correlations of internet flow characteristics. *Comput. Netw.*, 50(1):46–62, 2006.

[22] O. Lengál. Automated generation of processing elements for FPGA. Bachelor's thesis, FIT VUT Brno, 2008.

[23] Liberouter. The Liberouter project. `https://www.liberouter.org`.

[24] T. Martínek and M. Košek. NetCOPE: Platform for rapid development of network applications. In *Design and Diagnostics of Electronic Circuits and Systems, DDECS. 11th IEEE Workshop on*, pages 1–6, April 2008.

[25] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets:MIB-II. RFC 1213 (Standard), March 1991. Updated by RFCs 2011, 2012, 2013.

[26] M. Molina, A. Chiosi, S. D'Antonio, and G. Ventre. Design principles and algorithms for effective high-speed IP flow monitoring. *Computer Communications*, 29(10):1653 – 1664, 2006. Monitoring and Measurements of IP Networks.

[27] A. W. Moore, M. Crogan, and D. Zuev. Discriminators for use in flow-based classification. 2008.

[28] A. W. Moore and D. Zuev. Internet traffic classification using Bayesian analysis techniques. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–60, New York, NY, USA, 2005. ACM.

[29] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP Performance Metrics. RFC 2330 (Informational), May 1998.

[30] L. Polčák. Hardwarová akcelerace analýzy a extrakce položek z hlaviček paketů. Bachelor's thesis, FIT VUT Brno, 2008.

[31] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[32] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFCs 950, 4884.

[33] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[34] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[35] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer. Information Model for IP Flow Information Export. RFC 5102 (Proposed Standard), January 2008.

[36] J. Quittek, T. Zseby, G. Carle, and S. Zander. Traffic flow measurements within IP networks: Requirements, technologies, and standardization. *Applications and the Internet Workshops, IEEE/IPSJ International Symposium on*, 0:97, 2002.

[37] J. Quittek, T. Zseby, B. Claise, and S. Zander. Requirements for IP Flow Information Export (IPFIX). RFC 3917 (Informational), October 2004.

[38] G. Sadasivan, N.Brownlee, B. Claise, and J. Quittek. Architecture for IP Flow Information Export. Available online at `http://www.ietf.org/internet-drafts/draft-ietf-ipfix-architecture-12.txt`, September 2006.

[39] S. Saroiu, P. K. Gummadi, and S. D. Gribble. SProbe: A fast technique for measuring bottleneck bandwidth in uncooperative environments. In *IEEE Infocomm 2002*, 2007.

[40] IEEE Computer Society. IEEE Standard for Local and Metropolitan area networks – Virtual Bridged Local Area Networks, 2005.

[41] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 181–192, New York, NY, USA, 2005. ACM.

[42] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007.

[43] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf. `http://dast.nlanr.net/Projects/Iperf`.

[44] P. Špringl. Architektura programového vybavení monitorovací sondy na bázi toků. Master's thesis, FIT VUT Brno, 2009.

[45] M. Žádnik. Design of flow monitoring probe. Master's thesis, FIT VUT Brno, 2007.

[46] M. Žádnik, T. Pečenka, and J. Kořenek. Netflow probe intended for high-speed networks. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 695–698, Aug. 2005.

[47] X. Xiao, A. Hannan, and B. Bailey. Traffic engineering with MPLS in the internet. *IEEE Network Magazine*, 2000.

[48] Xilinx. Xilinx, Inc. `http://www.xilinx.com`.

Appendix A

Probe configurability and supported fields

As has been already discussed, one of the aims of the monitoring probe is configurability: it must meet specific requirements of a user. Thus, during the design and implementation phase, all components that must deal with the Unified Header and flow record structures have been implemented as configurable.

The specific definitions of flow record structure, protocol header fields and operations on these fields are derived from the IPFIX emerging standard. RFC 5102 [35] defines the so called *IPFIX information elements*, which are the basis for flow record field names.

The following table provides list of supported IPFIX information elements in current implementation. Full list of elements may be found in [35].

Also note, that some columns of the table are empty, which is not an error, but an indicator that the field with an empty column cannot be used as an item of that column, i.e. the field cannot be used in flow record or unified header.

Flow Record Field Name	**Unified Header Name**
ipVersion	uh_ipVersion
sourceIPv4Address	uh_sourceIPv4Address
sourceIPv6Address	uh_sourceIPv6Address
destinationIPv4Address	uh_destinationIPv4Address
destinationIPv6Address	uh_destinationIPv6Address
ipTTL	uh_ipTTL
protocolIdentifier	uh_protocolIdentifier
nextHeaderIPv6	uh_nextHeaderIPv6
ipDiffServCodePoint	uh_ipDiffServCodePoint
ipPrecedence	uh_ipPrecedence
ipClassOfService	uh_ipClassOfService
flowLabelIPv6	uh_flowLabelIPv6
fragmentIdentification	uh_fragmentIdentification
fragmentOffset	uh_fragmentOffset
fragmentFlags	uh_fragmentFlags
ipHeaderLength	uh_ipHeaderLength
ipv4IHL	Use ipHeaderLength instead
IP Header Fields	

Continued on next page ...

Flow Record Field Name	Unified Header Name
totalLengthIPv4	Use ipTotalLength instead
ipTotalLength	uh_ipTotalLength
payloadLengthIPv6	Use ipTotalLength instead
Transport Header Fields	
sourceTransportPort	uh_sourceTransportPort
destinationTransportPort	uh_destinationTransportPort
udpSourcePort	Use sourceTransportPort instead
udpDestinationPort	Use destinationTransportPort instead
udpMessageLength	uh_udpMessageLength
tcpSourcePort	Use sourceTransportPort instead
tcpDestinationPort	Use destinationTransportPort instead
tcpSequenceNumber	uh_tcpSequenceNumber
tcpAcknowledgementNumber	uh_tcpAcknowledgementNumber
tcpWindowSize	uh_tcpWindowSize
tcpWindowScale	uh_tcpWindowScale
tcpUrgentPointer	uh_tcpUrgentPointer
tcpHeaderLength	uh_tcpHeaderLength
icmpTypeCodeIPv4	Use icmpTypeCodeIPv4IPv6 instead
icmpTypeIPv4	Use icmpTypeCodeIPv4IPv6 instead
icmpCodeIPv4	Use icmpTypeCodeIPv4IPv6 instead
icmpTypeCodeIPv6	Use icmpTypeCodeIPv4IPv6 instead
icmpTypeIPv6	Use icmpTypeCodeIPv4IPv6 instead
icmpCodeIPv6	Use icmpTypeCodeIPv4IPv6 instead
igmpType	uh_igmpType
Sub-IP Header Fields	
sourceMacAddress	uh_sourceMacAddress
vlanId	uh_vlanId
destinationMacAddress	uh_destinationMacAddress
mplsTopLabelTTL	uh_mplsTopLabelTTL
mplsTopLabelExp	uh_mplsTopLabelExp
mplsTopLabelStackSection	uh_mplsTopLabelStackSection
mplsLabelStackSection2	uh_mplsLabelStackSection2
mplsLabelStackSection3	uh_mplsLabelStackSection3
mplsLabelStackSection4	uh_mplsLabelStackSection4
mplsLabelStackSection5	uh_mplsLabelStackSection5
mplsLabelStackSection6	uh_mplsLabelStackSection6
mplsLabelStackSection7	uh_mplsLabelStackSection7
mplsLabelStackSection8	uh_mplsLabelStackSection8
mplsLabelStackSection9	uh_mplsLabelStackSection9
mplsLabelStackSection10	uh_mplsLabelStackSection10
Min/Max Flow Properties	
minimumIpTotalLength	
maximumIpTotalLength	
minimumTTL	
maximumTTL	
IP Header Fields	

Continued on next page . . .

Flow Record Field Name	**Unified Header Name**
ipv4Options ipv6ExtensionHeaders tcpControlBits tcpOptions	
Flow Timestamps	
flowStartMicroseconds flowEndMicroseconds	
Per Flow Counters	
octetTotalCount octetTotalSumOfSquares packetTotalCount tcpSynTotalCount tcpFinTotalCount tcpRstTotalCount tcpPshTotalCount tcpAckTotalCount tcpUrgTotalCount	

Moreover the flow record description defines several other fields, not part of IPFIX. These fields are either compressed IPv4 and IPV6 packet properties merged into one flow record field or other special definitions which are not part of IPFIX, e.g. timing characteristics of a flow.

Flow Record Field Name	**Unified Header Name**
IPv4 and IPv6 Merged Fields	
sourceIpv4Ipv6Address	uh_sourceIpv4Ipv6Address
destinationIpv4Ipv6Address	uh_destinationIpv4Ipv6Address
icmpTypeCodeIPv4IPv6	uh_icmpTypeCodeIPv4IPv6
mplsTopLabelIPv4IPv6Address	uh_mplsTopLabelIPv4IPv6Address
Interval fields	
minimumInterval maximumInterval intervalTotalSum intervalTotalSumOfSquares	

Appendix B

CD medium

This electronic attachment contains all source codes for the hardware accelerated design of the probe and LaTeX source codes of this thesis. The CD can be found at the back side of the document.