Architecture of a Novel High Performance Traffic Capturing Device Based on the Intel IXP2400 Network Processor

D. Ficara*, S. Giordano*, F. Oppedisano*, G. Procissi* and F. Vitucci*
*Department of Information Engineering, University of Pisa
Address: Via Caruso 1, 56122 Pisa, Italy
Email: {domenico.ficara, s.giordano, f.oppedisano, g.procissi, f.vitucci}@iet.unipi.it

Abstract— **The extensive availability of cost effective commodity PC hardware pushed the development of flexible and versatile traffic monitoring software such as protocol analyzers, protocol dissectors, traffic sniffers, traffic characterizers and IDSs (Intrusion Detection Systems).**

The largest part of these pieces of software is based on the well known libpcap API, which in the last few years has become a *de facto* standard for PC based packet capturing.

Many improvements have been applied to this library but it still suffers from several performance flaws; these flaws are not generated by the software itself but by the underlying hardware bottlenecks.

In this paper we present the architecture and the implementation design of a new traffic monitoring device, implemented by an Intel IXP2400 network processor PCI-X card connected to a gigabit ethernet LAN hosting a cluster of common personal computers running any libpcap based application.

Since this is mainly an architectural work, only very preliminary experimental results are presented, while every design choice is justified from a theoretical point of view.

Index Terms— **Traffic monitoring, protocol analyzer, traffic classifier, network processor, packet capturing, traffic characterization**

I. Introduction and Motivations

In the last few years, the availability of flexible, easy to use and easy to customize network monitoring software, has proposed the PC as a suitable platform for network monitoring and testing. Application such as *tcpdump* [3], *wireshark* [1], *ntop* [2] etc., prove to be very effective and flexible for large a variety of monitoring tasks.

Most of these pieces of software are based on the well known *libpcap*API [3], which in the last few years became a *de facto* standard for PC based packet capturing.

Many improvements have been applied to this library [5] [6] but it still suffers from performance flaws; these flaws are not generated by the software itself but by underlying hardware bottlenecks [7] [8].

All these applications are often used together with high-end PCs to capture, analyze and characterize traffic from high-speed links; in all these cases, their main weakness is evident: low performance. Traffic traces produced by such a combination of hardware and software suffer from two types of uncertainty:

1) **Packet timestamps:** to sustain an high packet rate, the PC must drive interface cards by polling and this results into poor timestamp accuracy;
2) **Packet loss:** packet loss can be experienced if either the packet rate is too high and the host CPU cannot allocate/release memory for packets or if the system bus cannot keep the pace of the incoming data.

Moreover, in many cases, only off-line computing can be performed on incoming packets since no extra CPU power is left for on-line analysis (all the CPU time is used to capture packets) [7] [8].

This poor performance is mainly due to the lack of packet computing capabilities on the network interface cards which commonly equip commodity PCs: these interface cards are uncapable of either timestamping the arrival of a packet (avoiding interrupt latency) or to filter out unwanted packets (avoiding memory allocation/release for unwanted packets) or to feed the host PC with only a fragment of the packet instead of the entire one (avoiding system bus saturation).

The research activity described in this paper proposes an architecture to combine the flexibility of general purpose PCs (equipped with libpcap based applications) with the power of network processor cards of the Intel IXP2400 family (see section II). The result is a system capable of capturing packets on GigaEthernet links without loss and with good timestamp accuracy.

A. The Basic Idea and Issues

Figure 1 depicts a scenario where the two flow directions of a Gigabit Ethernet optical fiber are both split into two optical signals: the first signal is scattered to an output fiber while the second passes through the splitter. Hence, there are two output fibers, one for each direction.

This is the best available way to "copy" network traffic though some others are possible (e.g. configuring port mirroring on layer 2 network devices).

The output fibers of the splitter are connected to two of the three optical interfaces (see section II) of a Radisys ENP2611 Network Processor (NP) card, while a cluster of PC-based Linux boxes is connected to the third interface via a gigabit ethernet switch. PCs and NP are also connected via a standard 100BaseT Ethernet LAN (by using the *control interface* on the

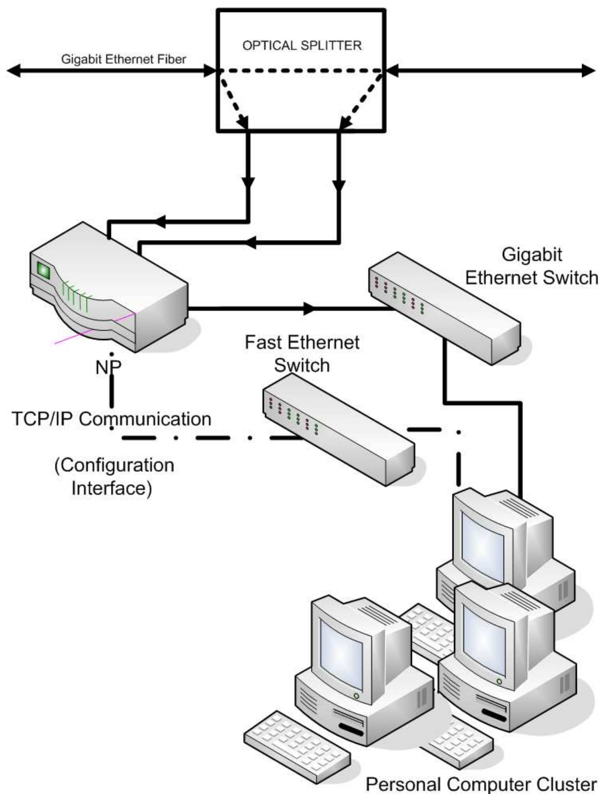

Fig. 1. Conceptual scheme of the monitoring system

NP) supporting a standard TCP/IP connection used to issue configuration commands from user interfaces; therefore every PC on the LAN can issue configuration commands to the NP via a client/server application (the server resides on the NP, while each PC runs an instance of the client).

This is currently the most general scenario in which it is possible to use such an architecture (for a three port scenario see section IV) but, in order to justify some of the following statements, we need to assume that only one of the two flow directions of the fiber (up-stream or down-stream) is really connected to the NP and monitored. In the following sections we will highlight the performance loss subsequent to the concurrent monitoring of both down and up-stream flow directions using more than one of the NP's interfaces.

Referring to this scenario, the basic idea behind the proposed architecture is to make the NP board performing at the wire speed (1Gbps: only up or down-stream) the following operations:

1) **Packet timestamping:** recording the arrival time of each packet in the standard UTC format;
2) **Packet classification and filtering:** selecting only those packets useful for the user and assigning each packet a unique flow identifier based on a rule set;
3) **Header striping:** getting only the necessary information (e.g. the first n bytes);
4) **Batch frame crafting:** collecting all this information in batch frames, each containing information of several (up to 19) packets;
5) **Sending batch frames to a commodity PC belonging to the cluster:** using the third port fiber of the NP board.

On the PC side, the batch frame is received and must be dissected and delivered to some monitoring application.

The main advantages of this architecture are:

1) **Timestamping accuracy**, in that it is performed by the NP card without the interrupt latency typical of a PC;
2) **Heavy CPU offload**, as unwanted packets are dropped at the NP level and are not delivered to any PC and since a pre-classification is performed on packets, bringing even more CPU offload (for example in flow identification).

At this stage, the main issue of this architecture would be the incompatibility between the proposed batch frame and all the available libpcap-based applications. Next sections describe the implementation design of the entire architecture made up of an NP-side timestamping and classification application and a PC-side kernel space abstraction layer which guarantees the compatibility with any libpcap-based application.

II. The Implementation Design

The IXP2400 is a fully programmable network processor, which implements a high-performance parallel processing architecture on a single chip suitable for processing complex algorithms, detailed packet inspection and traffic management at the wire speed. The IXP2400 architecture combines a high-performance Intel XScale core with eight 32-bit MEv2 microengines which cumulatively provide more than 5.4 giga-operations per second (capable of processing, namely, up to 3.6M packets per second). Each microengine has eight hardware-assisted (i.e.: zero-overhead context switch) threads of execution.

The Intel XScale core is a general-purpose 32-bit RISC processor (ARM Version 5 Architecture compliant) used to manage the NP, to handle exceptions and to perform slow path packet processing. The XScale processor and the whole set of μ-engines run at 600 MHz. Microengines provide the processing power necessary to perform *fast data path* tasks that would require very expensive high-speed ASICs. The IXP2400 NP is hosted by a third-party board. We adopt the RadiSys ENP-2611 board, equipped with 8 MB of (very fast) SRAM and 256 MB of (less fast) DRAM. This board provides three Gigabit Ethernet optical interfaces, one Fast Ethernet interface for remote control and supports MontaVista Linux operating system running on the XScale CPU. The board is plugged into a PCI-X slot of a host PC; this PCI connector is currently used only for power supply (no data communication takes place through it).

The main goal of the entire application is to accurately timestamp packets and to classify and extract a configurable portion of them within our IXP2400 Network Processor. The application is made up of an NP-side module and a PC-side module. A complete description of the whole application and its components will be elaborated upon in the next sections.

A. Network processor side: timestamping, classification, batch frame crafting and configuration

The NP side of our traffic monitor application reflects the IXP processor hierarchy: μ-engines are in charge of packet

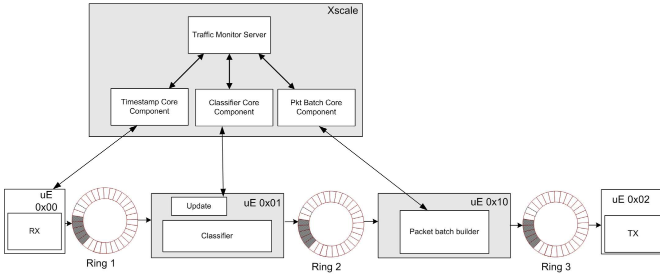

Fig. 2. Functional scheme of the entire NP-side application

timestamping and classification and batch frame crafting while the XScale deals with timestamp calibration, classification table setup and update and parameter reconfiguration. All the XScale functions are driven by a client/server application whose peer is the user interface residing on each of the cluster's PCs.

The entire NP-side application is depicted in Figure 2. In the figure, circles represent rings, which are on-chip circular FIFO queues (used for inter-μ-engine communication). The external rectangles represent processors. The internal rectangles represent the pieces of code that implement specific functions. In a μ-engine, these pieces are named μ-blocks, while at the XScale level they are called "core components".

1) μ-engines application scheme (timestamping, classification and batch frame crafting): referring again to Figure 2, white μ-engines contain the driver-blocks, directly provided by Intel and strictly hardware-dependent, dealing with low level functionalities.

The whole application can be summarized as follows.

The *RX* μ-engine retrieves packets from interface and puts them in Ring 1. For each packet, the arrival time (really the arrival time of the first mpacket, see below in this section), the entire length and the first n bytes of the packet (default=64) are recorded. The second μ-engine (uE 0x01) classifies the packets it receives from Ring 1 (by assigning them a *flow identifier (flowID)* between 0 and $2^{16} - 1$ or by simply dropping them) and sends them to the next μ-engine (uE 0x10) which copy all data buffers (each containing flowID, length, timestamp and the first n bytes of a packet) together to create a batch frame (the batch frame format is depicted in figure 3). Finally, the batch frame is passed to the Transmission μ-engine to be sent to one of the (max 32) PCs belonging to the cluster.

The batch frame header has the source address set as the MAC address of the outgoing interface, the destination address set as the MAC address of the correspondent cluster's PC and the type field set to an unused value (0x9000). As depicted in figure 3, the payload is build with a variable number of *packet digests* each made up of all the packet information (flowID, length, the n-bytes fragment and the arrival timestamp). The length of the fragment can be different among different flowIDs.

The code running on the 0x10 μ-engine (batch frame crafting) contains a table with the correspondence between flowID and the MAC address of the PC in charge of processing that flow. At a given time, the application maintains up to 32 batch frames (one for each PC of the cluster) and each

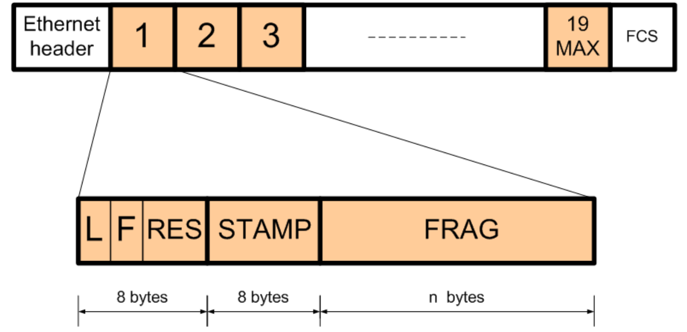

Fig. 3. Batch frame specification

packet digest is copied onto the batch frame correspondent to its flowID.

The timestamping operation consists of recording the arrival time of each packet. The arrival time is intended as the time t_a at which the first bit of the packet reaches the network interface. Unfortunately, packet reception (the action of retrieving packets from the wire to the CPU which is the first place where timestamping can be performed) is a compound operation (made up of many stages) in most network interfaces and RadiSys ENP-2611 board is not an exception. Then, in order to derive the timestamping accuracy of the system, we need to accurately examine what actually happens whenever a packet arrives at the board.

The Gigabit Ethernet interfaces of ENP2611 are controlled by a Sierra PM3386 and a PM3387 Gigabit MAC device (see figure 4). Those devices forward received frames to an FPGA Bridge connected to the MSF interface of the IXP2400. The MSF operates in POS-PHY Level 3 (aka SPI-3, aka PL3) mode and splits packets in fixed-sized chunks called *mpackets* (whose size is configurable as 64, 128 or 256 bytes). In order to avoid contention on the PM3386, in our application, one of the two interfaces connected to this chip is used for transmission, while the remaining one, together with the one connected to the PM3387 chip, is used for packet capturing.

At start-up time, all the *RX* μ-engine threads place themselves on a freelist (RX_FREELIST), thus stating they are ready to handle a new mpacket. Each time the MSF receives an mpacket, it awakes the first thread in the freelist and delivers the data to it. Then *RX* threads gather the set of incoming mpackets from MSF and merge them, thus reassembling original packets. When dealing with timestamping operation the main concern is the jitter of the delay each stage introduces (a fixed and known amount of delay between the real and measured time is not an issue since it can be subtracted to the measure). In the following, we shall try to show that upon receiving traffic from one interface, the delay between the arrival time and the timestamp operation is almost constant on the ENP2611.

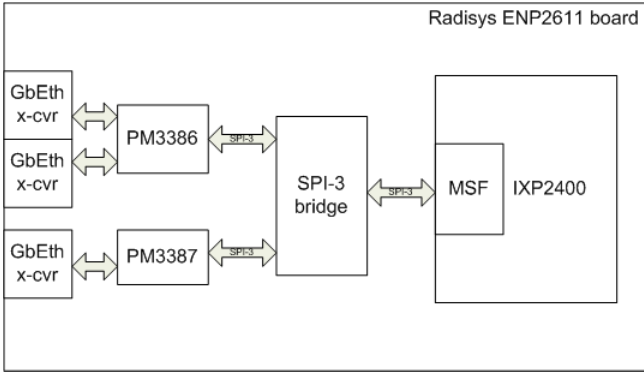

Fig. 4. Hardware packet receiving chain

Both the PM3386 controller and the SPI-3 bridge forward incoming frames as soon as a certain amount of bytes (we shall hereafter call it "forwarding threshold") is received. Forwarding threshold can be configured to 64, 128 or 256 bytes. Since the minimum packet size on Ethernet is 64 bytes, in order to avoid timestamp jitter due to different packet lengths, we set the "forwarding threshold" and the mpacket size to 64 bytes.

This way, the first thread in the RX_FREELIST is awaken at time t_x with a fixed delay from the arrival time t_a. The delay $t_x - t_a$ consists of the sum of three latencies corresponding to the three interfaces that data has to cross. The first delay is the time it takes to reach the "'forwarding threshold"' within the PM3386 and is simply given by $d_0 = 512\text{bits}/1\text{Gbps} = 0.512\mu s$. Then the PM3386 has to transfer data across the second interface toward SPI-3 bridge and it takes $d_1 = \frac{512\text{bits}/32\text{bits}}{104Mhz} \cong 0.154\mu s$. Finally, the SPI-3 interfaces operates at the same speed as the second one, thus adding an equal delay $d_2 = d_1$. Therefore, provided there is an available thread ready to timestamp the packet as soon as it arrives, the time lag $t_x - t_a = d_0 + d_1 + d_2 \cong 0.820\mu s$ is fixed and known.

We now have to prove that always there is such a thread. First of all, it is worth describing the operations performed by a thread of the *RX* μ-engine when it receives the signal that a packet arrived. Those operations are:

1) **Packet timestamping**
2) **Copying into the DRAM packet data and timestamp**
3) **Context switch**

In the first step, the μ-engine simply reads a timestamp counter (see below) and stores its value into an internal register; therefore, this step is very fast. In the second step, the μ-engine instructs the memory controller to write packet data and timestamp into the DRAM. These first two steps together take about 80 clock cycles to be executed. In the third step the μ-engine puts itself in an idle state until the memory executes the requested operations and switches to the first ready thread in the RX_FREELIST (context switch); when all the memory operations are completed, the thread is signaled by the memory hardware itself and can restart its operation. It is clear that, while the memory controller is executing the requested operations, the μ-engine can be used to perform other tasks by means of another thread. Hence, the total amaount of time necessary to timestamping a packet is $T_{proc} = 80ct + T_{lat}$, where ct stands for *cycle times* and T_{lat} is the memory latency. The worst case occurs whenever all packets are 65 bytes long: in this case, we have a 64 bytes long mpacket plus one extra 1 byte long mpacket. The total amount of time necessary to process this packet is the time necessary to process two mpackets (i.e. 2 times T_{proc}), while the interarrival time is 408ct. Hence, to have a steady state lossless packet processing, $T_{proc} \leq 204ct$ must hold. Experimentally, we observe that the RX μ-engine running this code can process up to about 3Mpps without loss, which proves that the above inequality holds.

Unfortunately this proves only that the mean value of the latency is such that the time budget is sufficient but it does not prove that there is always a ready thread since a spurious random memory latency could affect some timestamps. To avoid this random error, four of the eight thread of the RX μ-engine are assigned to the first interface, while the remaining ones are assigned to the second one. Being N_{th} the number of thread assigned to a given port, we can assume that there is always a ready thread in RX_FREELIST if RX processing delay is less than the minimum interarrival time (680 ns on Gigabit Ethernet) times the number of threads:

$$680\text{ns} \times N_{th} = 364\text{ct} \times N_{th} = 364\text{ct} \times 4 = 1456 \geq T_{proc} \quad (1)$$

Inequality (1) always holds since, experimentally, T_{proc} is always less than 200 clock cycles. As reported in the IXP2400 data sheet, it is possible to assume the signaling time to awake a *RX* thread to be very small. Thus, by using more than one thread per port, we make sure that the *RX* μ-engine threads can always receive the first mpacket of a packet with a fixed delay from the real arrival time.

As for packet batch creation, both the amount of data taken from each packet and the packet batch total size are configurable. Once the amount of data in the packet batch reaches the configured size, it is sent to the *TX* μ-engine. Moreover, a timeout is provided to make sure that non-full packet batches are transmitted if no more packets arrive.

Timestamp is provided by the use of 64-bit timestamp registers within the *RX* μ-engine. Such registers are increased by one every 16 clock cycles (we shall call it "'NP-tick"'). Then each packet is timestamped with a value given by $t_x - d_0 - d_1 - d_2 = t_x - 492\text{ct} = t_x - 31$ ticks.

What explained so far shows that the timestamp accuracy is sufficient for all the applications mentioned in section I; moreover it is worth to discussing the accuracy of the timer used for timestamping. As already mentioned, the timestamp counter (which is an internal register of the μ-engine) increases every 16 clock cycles. This means that each "tick" represent a time of $16/600\mu s = 0.03\mu s$. Just to quantify the goodness of such granularity, it is worth reminding that the most error sensitive application is traffic characterization; in this application the measure which has to be very accurate is the inter-arrival time of packets. Since the minimum inter-arrival time on a Gigabit Ethernet link is $0.68\mu s$, we obtain a maximum error of 4% using only one port, of 9% when using two ports and 13% when using three ports (the use of three ports has not been designed yet, see section IV). Users

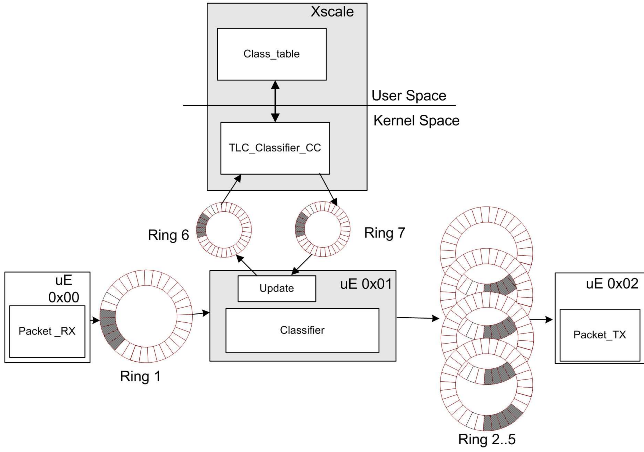

Fig. 5. Functional scheme of the classification application

involved in traffic characterization have to be aware of these granularity limitations.

If packets are concurrently captured from two interfaces, we have a timestamp error when two mpackets are presented to the SPI-3 chip by PM3386 and PM3387. An upper bound of the timestamping error is obtained in the worst case which takes place when two mpacket arrive exactly at the same time to the SPI-3. In this case one of the two mpackets has to wait $E_{max} = d_1 + 80ct \cong 0.287\mu s$ before being timestamped. Comparing this error with the minimum inter-arrival time we obtain an error of 42% which is much larger than the 9% due to the clock granularity ([1]).

We will not deal with the details of the classification application here, since it has already been published; readers interested in some classifier internals can look at [4].

2) Xscale application (configuration): The XScale processor handles the functionalities concerning timestamp calibration, classification table setup and update and parameters reconfiguration. Moreover, it cares of interfacing to the user-side (residing on the cluster PCs) by means of an instance of a client/server application for each PC of the cluster.

Timestamp calibration is the operation of resolving the correspondence between the μ-engine timestamp, which is a relative time, with the UTC time, which is an absolute time. This correspondence (initial value plus counter frequency) is passed to the PC-side application in order to allow a correct UTC timestamping (remember that the timestamps recorded in the batch frame are the values of a 64 bit counter integrated into the μ-engines; this number has no sense in the PC side).

As far as the classifier configuration concerns, a proper file is generated and passed to module Class_table (see Figure 5 and [4]), which builds the decision-making data structure. The XScale also manages the dynamic reconfiguration of the rule set: this is done through a message exchange between its *core components* and μ-engines [4]. Indeed, before the classifier table can be updated, the XScale must send a message to μ-engines in order to block any request to access to the classifier table during the update time. Meanwhile, packets can still be processed thanks to a cache table that stores flows information and classification results. This message exchange is triggered by relevant commands sent by the remote user interfaces residing on cluster's PCs.

The XScale application is also responsible of setting the desired number n of bytes to be captured for each packet belonging to a given flow and managing the flowID spaces (only flowIDs from 0 to $2^{16} - 1$ are possible) which have to be assigned to the various PCs without collisions.

[1]In a three port scenario, mpackets can experience contention not only on the SPI-3 bridge but also on the PM3386; therefore the computation of E_{max} in this case is slightly more complex.

B. PC side: compatibility abstraction layer and user interface

The PC-side application is composed by two components. The first is a Linux kernel module which implements a compatibility abstraction layer, while the second is a user space application built by a front-end (user interface) and a back-end which passes user's configuration commands to the kernel module (via ioctl system calls) and to the NP (via the above mentioned TCP/IP connection established on the Fast Ethernet control interface).

1) Kernel space – the compatibility abstraction layer: This module acts as a compatibility layer between the NP-PC communication protocol (which simply is the batch frame format), and the standard packet processing chain of the Linux kernel on which the libpcap API is based.

The module registers itself as a virtual "network layer" capable of processing ethernet frames with the type field equal to 0x9000.

The module also creates up to 2^{16} virtual interface cards mon0 to mon65535 (one for each flowID), then implementing an abstraction layer toward the system. Every time a batch frame is received by the kernel, it is steered to this layer which, in turn, extracts from its payload all the packets together with their timestamp and flowID. For every extracted packet digest, a new correctly timestamped packet is generated and transmitted on the virtual interface indexed by *flowID*.

Hence, a libpcap based application configured to monitor the interface e.g. mon5 (e.g. with the command tcpdump -i mon5) will see all (and only) those packets with flowID 5, as if it was connected directly to the fiber (to which actually the NP is connected).

Therefore, this layer makes it possible to instruct the NP to mark an arbitrary micro-flow with a specific flowID, and to analyze this flow by simply connecting an application, such as *wireshark*, to the corresponding virtual interface.

2) User Space – the user interface: The user interface is made up of a back-end capable of

- configuring the NP classifier via a TCP connection whose peer is the Xscale application;
- instructing the NP to capture the desired number n of bytes from every packet (via the TCP connection);
- reading from the same peer the association timestamp-UTC;
- configuring the abstraction layer via ioctl system calls.

The front-end module simply implements a user-interface from which the user can configure the entire system.

III. Current state of implementation and performance overview

The overall system described so far is in the process of actual implementation; currently it is at an intermediate stage of development.

To date, the classification application is in an almost mature development stage and some preliminary performance results show that:

- the system is able to support up to 15000 rules;
- a packet rate of 3 Mpps without loss and transfer delay values less than $20\mu s$ (regardless of traffic distribution and throughput) are guaranteed.

As far as the rule set configuration concerns, a PHP interface is created to allow rules insertion by network administrators.

The timestamping application and the kernel abstraction layer are in alpha development state, while the control infrastructure (TCP/IP connection and ioctl system call) are not yet implemented.

IV. Conclusion and future work

The paper proposes a packet capturing system architecture which combines the power of a network processor card with the flexibility of software based solutions; this brings wire speed capturing capacity to applications such as traffic monitors, intrusion detection systems, protocol analyzers.

This system has enough processing power to capture and classify up to 3Mpps, but such a packet rate can be reached by using more than one interface only; this, as already explained, introduces a loss of accuracy in the timestamps. It is up to the user to find the correct trade-off between the captured packet rate and the timestamp accuracy. In order to reduce the loss of timestamp accuracy, we are planning to redesign the application to assign one μ-engine per port, instead of one μ-engine for all the ports.

The entire system is significantly complex and not all of the components have been implemented yet. Nonetheless, some future refinements are planned for this architecture in order to overcome some design limitations.

The first limitation of the system is the use of a batch packet regardless of the fragment length configured by the user. Hence, if a user asks the system to capture the entire length of the flowing packets, the resulting batch packet is longer then the maximum ethernet frame every time a full size packet is captured. To overcome this problem we are planning to use oversized ethernet frames, exploiting the capability of modern Gigabit Ethernet NICs to handle such packets.

A second limitation is the use of a gigabit ethernet interface of the NP-board to send batch frames to the cluster; as a matter of fact, the number of interfaces on an NP board is one of the most important sources of cost and hence an interface is a resource to be saved. Therefore, a better way to performs operation would be, instead, to use the PCI-X bus (which currently only gives power supply to the board) to transmit the batch frames to the host PC and making it to forward them to the cluster via a less expensive gigabit ethernet PCI-X NIC. Only an experimental implementation of this solution will tell us whether it could be a viable solution from the performance point of view (will the host PC be able to forward all that packets without loss?). A variant of this solution will be the use of peer-to-peer PCI-X transactions in order to send the packets via the above mentioned PCI-X NIC connected to the host PC without involving the PC central CPU (thus solving the performance issue).

Finally, another limitation of the system is that it cannot currently classify and duplicate packets to more than one flow in order to send a given packet to more than one processing PC or application. A possible solution involves the upgrade of the classification application (which must be able of handling rules with multiple target flowIDs), of the batch frame format (and hence the PC side abstraction layer) to bring multiple flowIDs per packet digest and of the batch frame crafting application to make it capable of copying a fragment across many packet digests (possibly with different fragment lengths).

Despite the above mentioned limitations, both the theoretical basis of this architecture and the preliminary results obtained from its in-progress implementation make it promising as a general packet computing architecture to efficiently exploit the power of network processor hardware and the flexibility of software based monitoring solutions.

Acknowledgment

This work has been sponsored by the Italian MIUR research program MIMOSA.

References

[1] Wireshark protocol analyzer (was Ethereal), http://www.wireshark.org
[2] Ntop network traffic probe, http://www.ntop.org
[3] Lawrence Berkeley National Labs, tcpdump/libpcap, Network Research Group, http://www.tcpdump.org/
[4] S. Giordano, G. Procissi, F. Rossi, F. Vitucci. Design of a Multi-Dimensional Packet Classifier for Network Processors. Proc. of ICC 2006, June 2006
[5] P. Wood, libpcap-mmap, Los Alamos National Labs, http://public.lanl.gov/cpw/
[6] J. Mogul and K. Ramakrisnan, Eliminating Receive Livelock in an Interrupt-Driven Kernel, Proceedings of 1996 Usenix Technical Conference, 1996
[7] Luca Deri, Improving Passive Packet Capture:Beyond Device Polling Proceedings of SANE 2004, October 2004
[8] Luca Deri, Passively Monitoring Networks at Gigabit Speeds Using Commodity Hardware and Open Source Software, Proceedings of PAM 2003, La Jolla, April 2003