Scalable network monitors for high-speed links: a bottom-up approach

Trung Nguyen[†], Mihai Cristea[†], Willem de Bruijn[*], Herbert Bos[*]
[†]Leiden Universiteit, The Netherlands
{tnguyen,cristea}@liacs.nl
[*]Vrije Universiteit Amsterdam, The Netherlands
{wdb, herbertb}@few.vu.nl

***Abstract*— Monitoring traffic on high-speed links using commodity hardware is difficult due to relatively slow buses and memories. It is possible to alleviate the burden on these resources by pushing down packet processing to programmable NICs. Until now, however, the use of such cards for monitoring by network administrators has not been a practical solution, because programming the cards is too complex. For this purpose, we introduce** NIC-FIX**, a monitoring framework for network processors that scales to high link rates and is easy to use.**

I. Introduction

Packet handling in modern workstations works well for slow, sub-gigabit speeds but fails badly at higher rates. While some operating systems fare a little better than others, this statement is true irrespective of one's choice of operating system [1]. The problem is caused by a combination of hardware and software bottlenecks.

Memory and peripheral bus technologies struggle to keep up with backbone link rates. Even if a workstation does manage to get all packets across the bus in host memory and from memory in the CPU, inefficient packet handling by the OS still makes it difficult to process packets at high speeds. The problems are commonly rooted in the overhead of interrupt handling, context switching and packet copying [2]. As it stands, we conclude that common workstations with current hardware and software configurations are not suitable for high-speed network monitoring. At the same time, the need for affordable network monitors is growing, e.g., for security, traffic engineering, SLA monitoring, charging, and other purposes.

In this paper we present NIC-FIX, an implementation of the Fairly Fast Packet Filter (FFPF) [3] network monitoring architecture on network cards with Intel IXP1200 network processors (NPU) [4]. FFPF has been used as a codebase for implementing the Monitoring API (MAPI) developed within the European Scampi Project [5]. Its architecture can be described as 'bottom-up' in that packets are handled at the lowest processing level and few packets percolate to higher levels. Moreover, higher levels only take action when prompted to do so by the lower layers. This is a well-known approach, e.g., in router design [6].

NIC-FIX forms the lowest level of the FFPF hierarchy. It allows us to deal with common hardware bottlenecks by offloading packet processing to the network card when possible, thus reducing strain on the memory and peripheral buses. Complementing the framework we also introduce FPL-2, a novel filter language that is more powerful than existing ones such as BPF or its Linux cousin LSF [7] and has been implemented on top of NIC-FIX.

The rest of this paper is laid out as follows: in Section II we give an overview of the FFPF architecture. Section III discusses the specifics of NIC-FIX, while Section IV is devoted to the FPL-2 filter language. The software is then evaluated in Section V. Related work is discussed throughout the text and summarized in Section VI. Conclusions are drawn in Section VII.

II. FFPF overview

We will now summarize the relevant aspects of the FFPF architecture. A more detailed discussion can be found in [3]. FFPF was designed to meet the following challenges: (1) monitor high-speed links and scale with future link rates, (2) offer more flexibility than existing packet filters, and (3) provide a migration path by being backward compatible with existing approaches (notably `pcap`-based applications [7]).

A. Flows

A key concept in FFPF is the notion of a *flow*. Flows are simply defined as a subset off all network packets. This definition is broader than the traditional notion of a flow (e.g., a 'TCP connection') and encompasses for instance all TCP SYN packets or all packets destined for the user with UID 0. To accommodate for such diverse flows, FFPF, instead of specifying filters itself, allows for varied selection criteria through extensions. This makes it more versatile than traditional flow accounting frameworks (e.g. NetFlow or IPFIX [8]). Furthermore, FFPF filters can be interconnected into a graph structure similar to that of the Click [9] router for even more fine-grained control. A filter embedded in such a graph is called a *flowgrabber*.

B. Grouping

The *flowgroup* constitutes a second key concept in FFPF. Flowgroups allow multiple flows to share their resources. As resource sharing poses a security hazard group membership

is decided by an application's access constraints. Network packets can be shared safely between all applications in a single flowgroup. Whenever a packet is accepted by one or more filters in a flowgroup, it is placed in a circular packet buffer (`PBuf`) only once, and a reference to this packet is placed in the individual filters' index buffers (`IBuf`). In other words, there is no separate packet copy per application. As buffers are memory mapped, there is no copy from kernel to userspace either.

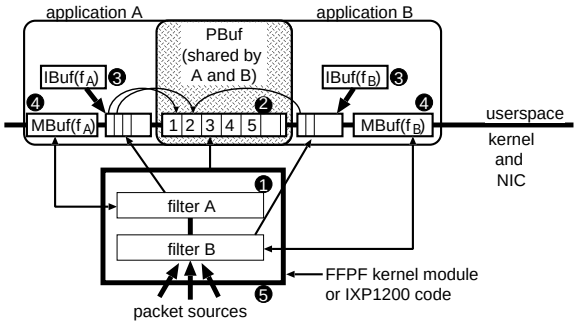

Fig. 1. The FFPF architecture

Figure 1 shows FFPF in its most basic form. Flows are captured by user-supplied filters ①. Corresponding to each of the filters are three buffers: the main packet buffer `PBuf` ②, a filter-specific index buffer `IBuf` ③, and a filter-private memory area called `MBuf` ④ that can be used for keeping state. For instance, a packet counting filter may use `MBuf` to store its counter. Packets enter the system on one or more 'packet sources' ⑤. Currently, three packet sources are implemented: two for regular NICs and one for NIC-FIX. In this paper, we focus on the latter.

C. Flowgraphs

In the full FFPF architecture, filters written in different languages may be mixed and matched in a directed acyclic flow graph. An example is shown in Figure (2.a). In the example, two flows A and B are captured, both of which contain only webtraffic. Flow A consists of worm signatures within this webtraffic while flow B is used to count the number of IP fragments in all web traffic.

The example illustrates the following points. First, the utilized filters generate both simple *statistics* (a counter) and more complex *state* (sets of IP addresses). Second, flow A is computationally expensive, performing a full payload scan (which is impossible with existing filter languages like BPF). Third, the flow graphs of the two flows are combined for efficiency, by executing the first filter only once. Whenever a new flow is created, FFPF automatically checks whether it can be embedded in an existing flow graph. Fourth, different languages are mixed. Fifth, as fragmentation is rare and few packets contain a worm, in the common case there is no need for the monitoring application to be scheduled at all (reducing context switches).

Building these complex graphs is no more difficult in FFPF than creating a complex program by connecting simple tools using UNIX pipes. For example, pronouncing the construct '`->`' as 'connects to' and '`|`' as 'in parallel with', the command in Figure 3 captures two different flows. The top flow specification indicates that the filter should capture packets at device `eth0`, pass them to a sampler that captures one in two packets and requires four bytes of `MBuf`. Next, sampled packets are sent both to a bytecount function and to an FPL-1 filter (a predecessor of FPL-2). The bytecount function does not take any arguments, but requires eight bytes of `MBuf` to store the result. The FPL-1 filter executes a user-specified filter expression (indicated by "..."), and in this example requires no `MBuf`. The result of these last two functions is forwarded to a special function `accept`, which delivers the packets to the application. In this (silly) example, two filters may pass the same packets via the same `accept`, so applications may receive the same packets twice. The second flow has two filters in common with the first expression, but now the packets are forwarded to another FPL-1 filter, and all packets passing this filter are accepted.

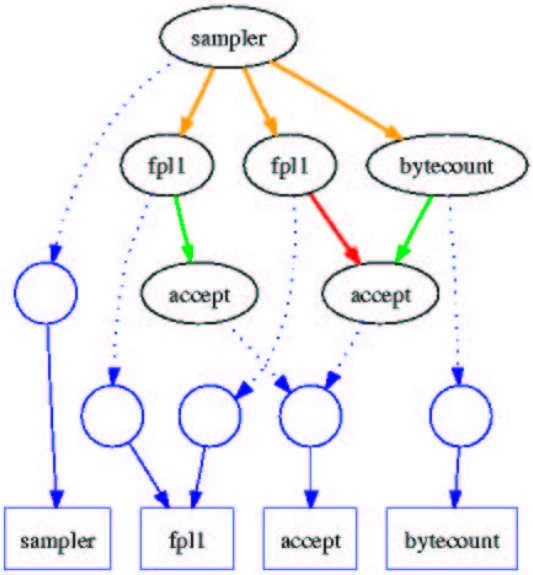

Fig. 4. autogenerated flowgraph image

As a by-product, FFPF generates a graphical representation of the entire flow-graph. For example, Figure 4 shows the two flows defined above (minus the devices). For simplicity, we have shown a very minimal configuration in which just four filter classes are present and everything happens in a single level of the processing hierarchy.

The active elements, *flowgrabbers*, are shown as ovals while the edges between them denote packet flows. The location of independent filters themselves are shown as circles. Finally, the rectangles at the bottom show filterclasses, blueprints from which the filters are instantiated. These seemingly vague distinctions help to reduce work duplication. In the example,

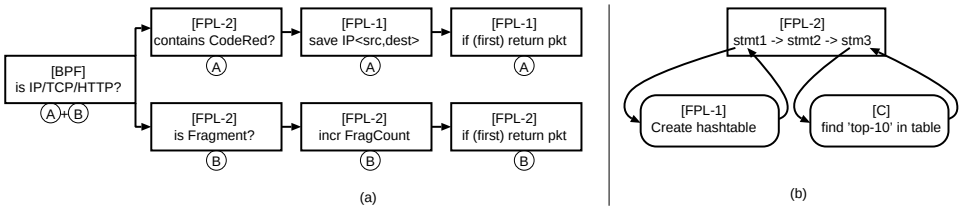

Fig. 2. (a) combining different languages in two flows (A and B), (b) calling external filters functions from a single flow

```
ffpf_flow "(dev,eth0) -> (sampler,2,4) -> (bytecount,,8) | (fpl1,"...") -> accept" \
          "(dev,eth0) -> (sampler,2,4) -> (fpl1,"...") -> (accept)
```

Fig. 3. FFPF from the command line: two partially overlapping flow graphs

for instance, two filters are seen to share a filterclass 'fpl1' which means that the same interpreter can be called with different expressions. Also, two flowgrabbers can rely on a single filter, as is the case with the 'accept' grabbers.

D. Buffer management

Buffer management concerns how readers and writers synchronize and share their buffers. FFPF supports two synchronization modes: 'slow reader preference' (SRP) and 'fast reader preference' (FRP). SRP corresponds to 'traditional' buffer management in which new packets are dropped if the buffer is full. While convenient, the disadvantage is that one tardy application that fails to read the (shared) `PBuf` at a sufficiently high rate, causes packet loss for the entire group. FRP, in contrast, simply keeps writing regardless of whether the buffer is full. The trick is that it enables applications to check *a posteriori* whether the packets they just processed have been overwritten. For efficiency, applications in both SRP and FRP may process their packets in *batches* (e.g. 1000 packets at a time) in order to minimise context switches. Details about SRP and FRP are provided in [3].

Filters may read the `MBuf` of other filters in their flow group. In case the same `MBuf` needs to be written by multiple filters, the solution is to use function-like *filter calls*, rather than pipe-like *filter connection* discussed so far. For filter call semantics, a filter is called *explicitly* as an external function by a statement in an FPL expression which will execute the target filter with the calling filter's `IBuf` and `MBuf`. An example is shown in Figure (2.b), where a first filter call creates a hash table with counters for each TCP flow, while a second filter call scans the hash table for the top-10 most active flows. Both access the same memory.

E. Processing stack

FFPF consists of a 3-level processing stack, executing (1) in userspace, (2) in kernelspace, and (3) on the NIC. Figure 5 shows the complete stack including userspace support libraries such as libpcap. Not all levels have to be used. For instance, FFPF supports `pcap` applications without modification, by using solely the lower two levels. Similarly, it can run on existing commodity hardware without processing at the NIC.

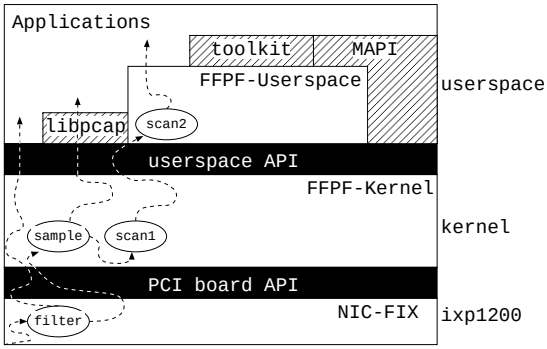

Fig. 5. The NIC-FIX software architecture with 3 example flows

When users instantiate a composite flowgraph such as shown in Figure 2, it remains transparent to the application at which levels its filters are executing. FFPF automatically finds the most appropriate level in the hierarchy. The rule is that a filter f is instantiated at the lowest level that supports the function and that is equal to or higher than the levels of all f's immediate predecessors in the flow definition.

III. NIC-FIX: FFPF ON THE IXP

A. Network Processors

Communication and memory bus bottlenecks can be circumvented by moving packet processing to the NIC. For such tasks special purpose devices are often considered the best solution as these are dedicated and fast. However, such ASIC based boards are too inflexible for fast changing tasks like network monitoring (e.g. worm signature checking). Also, their high costs interfere with FFPF's goal of using commodity hardware.

A solution that shares some of the advantages of dedicated devices with the cost-effectiveness of commodity hardware is the network processor (NPU). Several NPU platforms have been created, such as IXP by Intel, PowerNP by IBM and C-Port by Motorola. Network processors are dedicated pieces of equipment consisting of high-speed memory and fast interconnects. Contrary to high-end solutions, however, they do not contain preprogrammed ASIC's, but use reprogrammable 'micro' processors.

The Radisys ENP 2506 network card that was used to implement NIC-FIX is displayed in Figure 6. For input, the board is equiped with two 1Gpbs Ethernet ports ①. The card also contains a 232 MHz Intel IXP1200 network processor with 8 MB of SRAM and 256 MB of SDRAM ② and is plugged into a 1.2 GHz PIII over a 32/66 PCI bus ③. The IXP is built up of a single StrongARM processor running Linux and six independent RISC processors, known as μEngines, running no operating system whatsoever. Each μEngine supports four *threads* that have their own program counters and register sets and support zero-cycle context switches. Each μEngine has its own code 1K instruction store.

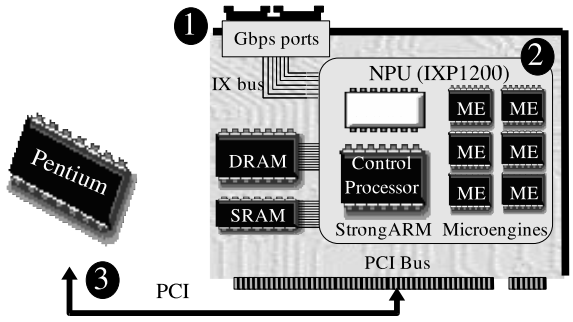

Fig. 6. Main components of the ENP-2506 board

B. NIC-FIX *software architecture*

Packets arriving at the IXP's network ports are filtered by the μEngines. A packet that passes the filters is stored in SDRAM on the card and a reference is passed to the host. Whether the packet itself is transferred to host memory depends on the configuration (as will be discussed in Section III-E). We do not suggest that complex processing, such as full payload pattern matching, is likely to performed on the μEngines (although this is not precluded). Rather, we propose to use the card mainly as a *prefiltering* stage.

Per network interface a single μEngine is responsible for receiving packets. The remaining μEngines are available to execute application-specific filter programs.

C. Filters

The basic structure of the programs running on the μEngines is as follows. For each μEngine, NIC-FIX provides boilerplate code in μEngine C, a subset of the C language. This boilerplate code contains a standard main() processing loop and a slot for the application-specific filter function.

In principle, users may define the entire body of their filter function directly in μEngine C. However, in practice this is complex. That is why NIC-FIX also exports a higher-level approach, whereby filters are written in a *special purpose* filter language (discussed in Section IV) which can be precompiled to μEngine C. The intermediate code combined with the boilerplate is then compiled to object code. This scheme closely resembles that of hardware plugins [10]).

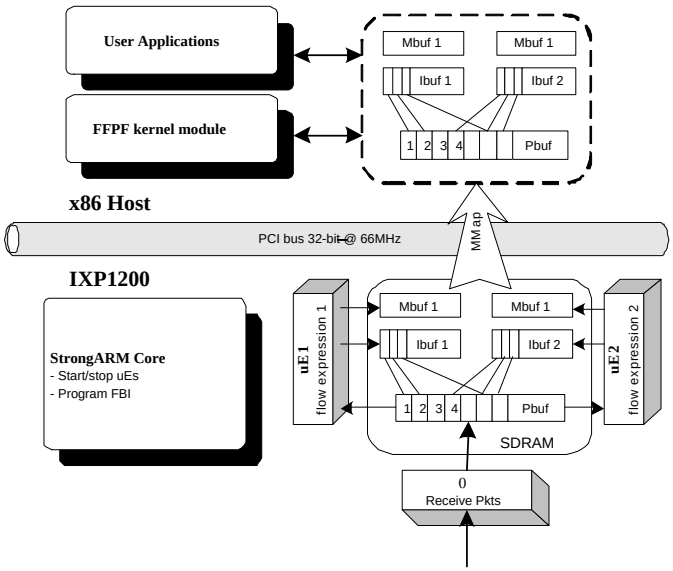

Fig. 7. NIC-FIX software organisation: the memory on the card is mmapped to the host (as indicated by the dashed 'ghost image' of the buffers).

D. On-board filtering

When a packet arrives at a network port its dedicated receiver μEngine copies the packet to the next available slot in `PBuf`. The remaining μEngines each execute a single filter. All four threads on a μEngine execute the same filter, but on different packets. If the packet matches the filter, the μEngine places a reference in the flow's `IBuf`. Otherwise, it is ignored. A packet is considered interesting when the user-specified filtering function returns a non-zero result.

The `PBuf` resides in the NIC's SDRAM and consists of static slots sufficiently large for any Ethernet frame. As shown in Figure 7, all three NIC-FIX buffers are memory mapped at the host processor.

E. Communication and Coordination

The StrongARM is only used for control tasks, which include μEngine initialization and control, input queue coordination and memory management.

Communication between the IXP1200 and the host computer is implemented simply by mapping the board's SDRAM memory to the host over the PCI bus. The IXP1200 supports sophisticated DMA channels which can move data from SDRAM to the PCI and can be accessed directly from the μEngines and the StrongARM. However, we are so far unable to use these. Instead, polling is used to communicate between host and IXP.

NIC-FIX supports three modes with respect to packet copying: (1) never ('zero-copy' mode), (2) always ('copy once' mode), and (3) *as needed* (a mode known as 'regular'). In *zero-copy* mode, packets remain on the card, no matter where they are accessed. In contrast, *copy-once* always incurs a (single) copy, as soon as a μEngine classifies the packet as 'interesting'. Which mode is better depends on whether code on the host accesses the packet data frequently. If so, zero

copy is slow, as all reads incur a PCI round trip time. On the other hand, if packets are hardly touched, zero copy is cheap.

A hybrid between the two extremes is formed by 'regular' mode. In this case, the packet is not moved from the card until it is *queued* for userspace applications, i.e. packets *may* well remain in the card's SDRAM for their entire 'lifetime'.

IV. The FPL-2 compiler

In NIC-FIX, users express their filters in FPL-2 (the FFPF packet language 2). FPL-2, summarized in Table I, is a new language that compiles to fully optimised μEngine object code. It supports all common integer types (signed and unsigned bits, nibbles, octets, words and double words) and allows expressions to get hold of any field in the packet header or payload in a friendly manner. Moreover, offsets in packets can be variable, i.e., determined by arbitrary expressions. For convenience, an extensible set of macros allows the use of shorthand for packet fields, e.g. instead of asking for bytes nine and ten to obtain the IP header's protocol field, a user may abbreviate to '`IP_PROTO`'. We briefly explain the constructs that may not be intuitively clear.

operator-type	**operator**
Arithmetic	`+, -, *, /, %, --, ++`
Assignment	`=,*=, /=, %=, +=, -=` `<<=, >>=, &=, ^=, |=`
Logical/Relational	`==, !=, >, <, >=, <=,` `&&, ||, !`
Bitwise	`&, |, ^, <<, >>`
statement-type	**operator**
if/then/else	IF (expr) THEN stmt1 ELSE stmt2 FI
for()	FOR (initialize; test; update) stmts; BREAK; stmts; ROF
external function	EXTERN(filter, input, output)
hash()	HASH(start_byte, #bytes, tablesize)
return value	RETURN (val)
Data type	**syntax**
Register n	R[n]
Memory location n	M[n]
Packets access:	
- byte $f(n)$	PKT.B[$f(n)$]
- word $f(n)$	PKT.W[$f(n)$]
- dword $f(n)$	PKT.DW[$f(n)$]
- bit m in byte n	PKT.B[n].U1[m]
- byte m in word n	PKT.W[n].U8[m]
etc.	(many options, including macros)

TABLE I
FPL-2 language constructs

- For resource safety, `FOR` loops are restricted with a predetermined upperbound on the number of iterations. The `BREAK` instruction, allows one to exit the loop 'early'.
- The concept of an 'external function' implements the filter call semantics mentioned earlier (and illustrated in Figure (2.b)). In FPL-2, an external function is called using the `EXTERN` construct. The input and output arguments allow parameters to be passed to the 'function' call. For instance, '`EXTERN(foo,x,y)`' calls filter `foo`, which reads its arguments (if any) from offset x in the filter's `MBuf` and produces output, if any, in the same buffer at offset y. In addition, the callee may return an integer result. External functions are very useful as they allow users to call efficient hardware or software implementations of computationally expensive functions (e.g., checksum calculations, hashes, or pattern matching engines).
- FPL accesses the filter's `MBuf` by means of the assignment operator. For instance, one may assign the content of a memory location to a register, perform a set of calculations, and then assign the value of the register back to memory. Examples of `MBuf` usage in FPL-2 are shown in Figure 8. For instance, in Figure (8.B), the code keeps track of how many packets were received on each TCP connection (assuming for simplicity that FPL-2's built-in hashing function is unique for each live TCP flow).

The packet language hides most of the complexities of the underlying hardware. For instance, users need not worry about reading packet data in a μEngine's SDRAM read registers first, before accessing it. The compiler generates boilerplate code to make such transfers transparent. Similarly, accesssing bytes or words in memories that are not byte addressable, are handled automatically by the compiler. If needed for efficiency, however, users may *choose* to expose some of the complexity. For instance, it is possible to declare additional memory arrays (besides `MBuf`) explicitly in Scratchpad, SRAM, or SDRAM.

As it is difficult to compete both in writing efficient code optimizers, and in providing an efficient packet processing environment, we have chosen to exploit the existing optimizer of the Intel μEngine compiler. For this reason, we reworked the kernel version of the FPL-2 compiler in such a way that it generates the `filter_impl()` plug-in discussed in Section III-B. As the target language of the new FPL-2 compiler is μEngine C, this file is subsequently wrapped in NIC-FIX template code and compiled and optimised by Intel's μEngine C compiler. Finally, the code is loaded on the card.

V. Experimental analysis

NIC-FIX was designed to scale with link rates, but as we do not have a 40 Gbps testbed, we evaluate the architecture with the setup of Section III-A, while monitoring a gigabit link. We deliberately 'under-engineered' the workstation to *not* be able to handle linkrate.

A. On-board processing

An important constraint for monitors is the cycle budget. At 1 Gbps and 100 byte packets, the budget for four threads processing four different packets is almost 4000 cycles. As an indication of what this means, Table II shows the overhead of some operations. Note that these results include all boilerplate (e.g., transfers from memory into read registers and masking).

Without NIC-FIX the maximum rate at which we can monitor the network is 602 Mbps for maximum-size packets, not nearly line-rate. To evaluate NIC-FIX, we execute the three filters shown in Figure 8 on various packet sizes and measure throughput. Only A is a 'traditional' filter. The other two gather information about traffic, either about the activity

(A) filter packets:

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

(B) count TCP flow activity:

```
// count number of packets in every flow,
// by keeping counters in hash table (of size 1024)
IF (PKT.IP_PROTO == PROTO_TCP) THEN
   R[0] = Hash(26,12,1024); // hash over TCP flow fields)
   // increment the pkt counter at this position
   M[ R[0] ]++;
FI
```

(C) count all occurrences of a character in a UDP packet:

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

Fig. 8. Example FPL-2 filters

in every flow (assuming the hash is unique), or about the occurrence of a specific byte. Note that the hashfunction used in B utilizes dedicated hardware support. The results are shown in Figure 9. We implemented three variations of filter C. In $C1$ the loop does not iterate over the full packet, just over 35 bytes (creating constant overhead). In $C2$, we iterate over the full size, but each iteration reads a new quadword (8B) rather than a byte. $C3$ is Figure 8 without modifications.

Description	Value
R[0] = HASH(26,12,256)	200 cycles
R[0] = PKT.B[0]	110 cycles
R[0] = PKT.W[0]	120 cycles

TABLE II

APPROXIMATE OVERHEAD OF SOME OPERATORS

Above a packet size of 500 bytes, NIC-FIX can process packets at line rate for A, B and $C1$. This means that if 10% of the traffic consisted of packets that match filter A, the prefiltering in NIC-FIX ensures applications like `tcpdump` would also handle link rate.

For smaller packets, filters $C1-3$ are not able to process the packets within the cycle budget. Up to roughly 165.000 pps $C1$ still achieves throughputs of well above 900 Mbps. Beyond that, the constant overhead cannot be sustained. $C2$ and $C3$ require more cycles for large packets and, hence, level off sooner. This suggests that simple prefilters that do not access every byte in the payload are to be preferred. This is fine, as the system was intended precisely for that purpose.

Just as for the C filters, throughput also drops for the simple filters A and B when processing smaller packets. However, these drops occur for a different reason, namely because the *receiving* μEngine simply cannot keep up.

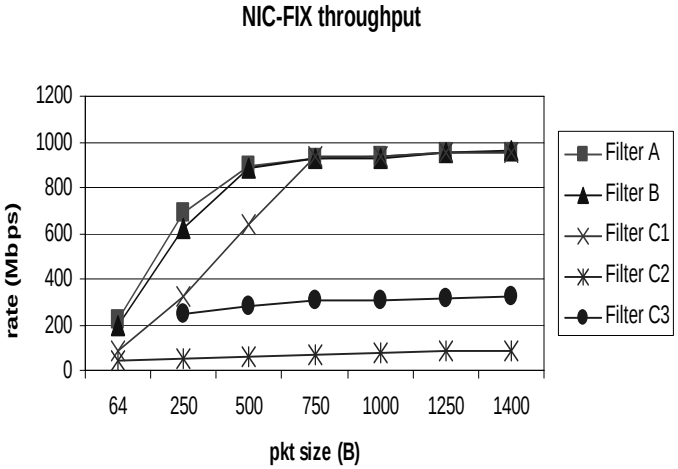

Fig. 9. Throughput for different NIC-FIX filters

B. Host communication

Figure 10 compares the three copy modes ('zero', 'once' and 'regular') for various scenarios: (1) the packet is *referenced*, but dropped even before entering kernelspace FFPF, (2) the packet is *dropped* in kernelspace, and (3) the packet is *accepted* and sent up to userspace (incurring a copy in regular mode). While sending at link rate, we measured how many packets arrived at the host. It is clear for these particular tests that overall zero-copy outperforms the other modes. We stress, however, that this performance depends on the application.

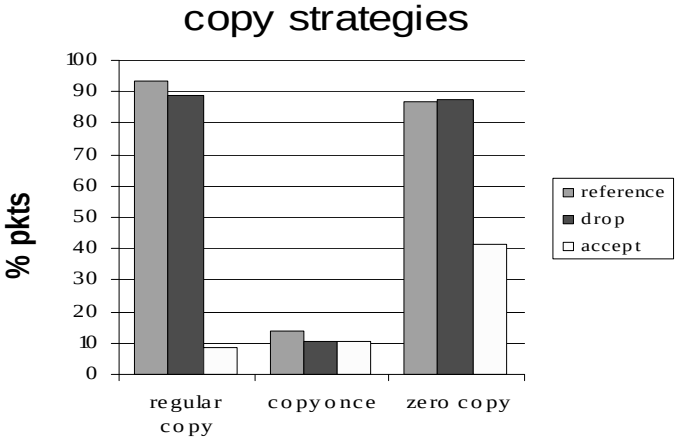

Fig. 10. Performance in % of packets received with different copy modes

VI. RELATED WORK

Many tools for monitoring are based on BPF in the kernel [7]. Filtering and processing in network cards is also promoted by some Juniper routers [11] and the Scout project [12]. Both lack features introduced in NIC-FIX such as extended languages, in-place packet handling and flow grouping. The SCAMPI architecture also pushes processing to the NIC [5]. It assumes the hardware can write packets immediately in host memory (e.g., by using DAG cards [13]) and implements access to packet buffers through a userspace

daemon. SCAMPI does not support user-provided external functions, powerful languages such as FPL-2 or complex filtergraphs. In programming IXPs, recent projects that have drawn attention are NPClick [14] and netbind [15]. While these project introduce interesting programming models they are not designed for monitoring. Support for high-speed traffic capture is provided by OCxMon and the high-speed monitor developed by Sprint [16], [17], both of which use DAG cards to capture packets at link rate.

VII. Conclusions

This paper presented NIC-FIX, an implementation of the FFPF packet filtering architecture that enables administrators to monitor network traffic at line rates by offloading computational tasks to the NIC. The experimental results showed that NIC-FIX outperforms traditional packet filters and allows a common PC to cope with line rate.

Acknowledgements

This work was supported by the EU SCAMPI project IST-2001-32404, while Intel donated the network cards.

References

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

[2] Jeffrey C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[3] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis, "FFPF: Fairly Fast Packet Filters," in *OSDI'04 (accepted for publication)*, San Francisco, CA, December 2004.

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

[5] Michalis Polychronakis, Evangelos Markatos, Kostas Anagnostakis, and Arne Oslebo, "Design of an application programming interface for ip network monitoring," in *IEEE/IFIP Network Operations and Management Symposium*, Seoul, Korea, April 2004.

[6] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb, "Building a Robust Software-Based Router Using Network Processors," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Alberta, Canada, October 2001, pp. 216–229.

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

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

[9] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek, "The click modular router," in *Symposium on Operating Systems Principles*, 1999, pp. 217–231.

[10] J.S.Turner, J.W.Lockwood, and E.L. Horta, "Dynamic hardware plugins (dhp): exploiting hardware for high-performance programmable routers," *Computer Networks*, vol. 38, no. 3, pp. 295–310, Feb. 2002.

[11] Tom M. Thomas, *Juniper Networks Reference Guide: JUNOS Routing, Configuration, and Architecture*, chapter Juniper Networks Router Architecture, Number ISBN: 0201775921. Addison Wesley Professional, January 2003, `http://www.awprofessional.com/title/0201775921`.

[12] Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, and Per Gunningberg, "Silk: Scout paths in the linux kernel, tr 2002-009," Tech. Rep., Department of Information Technology, Uppsala University, Uppsala, Sweden, Feb. 2002.

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

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

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

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

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