

FPL-3: towards language support for distributed packet processing

Mihai-Lucian Cristea¹ and Willem de Bruijn² and Herbert Bos³

cristea@liacs.nl, +31715277037, Leiden University, Niels Bohrweg 1, 2333CA
wdb@few.vu.nl, +31204447790, Vrije Universiteit Amsterdam, De Boelelaan 1081HV
herbertb@cs.vu.nl, +31204447746, Vrije Universiteit Amsterdam, De Boelelaan 1081HV,
The Netherlands

Abstract. The FPL-3 packet filtering language incorporates explicit support for distributed processing into the language. FPL-3 supports not only generic header-based filtering, but also more demanding tasks, such as payload scanning, packet replication and traffic splitting. By distributing FPL-3 based tasks across a possibly heterogeneous network of processing nodes, the NET-FFPF network monitoring architecture facilitates very high speed packet processing. Results show that NET-FFPF can perform complex processing at gigabit speeds. The proposed framework can be used to execute such diverse tasks as load balancing, traffic monitoring, firewalling and intrusion detection directly at the critical high-bandwidth links (e.g., in enterprise gateways).

Key words: High-speed packet processing, traffic splitting, network monitoring

1 Introduction

There exists a widening gap between advances in network speeds and those in bus, memory and processor speeds. This makes it ever more difficult to process packets at line rate. At the same time, we see that demand for packet processing tasks such as network monitoring, intrusion detection and firewalling is growing. Commodity hardware is not able to process packet data at backbone speeds, a situation that is likely to get worse rather than better in the future. Therefore, more efficient and scalable packet processing solutions are needed.

It has been recognised that parallelism can be exploited to deal with processing at high speeds. A network processor (NP), for example, is a device specifically designed for packet processing at high speeds by sharing the workload between a number of independent RISC processors. However, for very demanding applications (e.g., payload scanning for worm signatures) more power is needed than any one processor can offer. For reasons of cost-efficiency it is infeasible to develop NPs that can cope with backbone link rates for such applications. An attractive alternative is to increase scalability by exploiting parallelism at a coarser granularity.

We have previously introduced an efficient monitoring framework, Fairly Fast Packet Filters (FFPF) [?], that can reach high speeds by pushing as much of the work as possible to the lowest levels of the processing stack. The NIC-FIX architecture [?] showed

how this monitoring framework could be extended all the way down to the network card. To support such an extensible programmable environment, we introduced the special purpose FPL-2 language.

In this paper, we exploit packet processing parallelism at the level of individual processing units (NPs or commodity PCs) to build a heterogeneous distributed monitoring architecture: NET-FFPF. Incoming traffic is divided into multiple streams, each of which is forwarded to a different processing node (Fig. 1). Simple processing occurs at the lower levels (root nodes), while more complex tasks take place in the higher levels where more cycles are available per packet. The main contribution of this paper consists of a novel language that explicitly facilitates distribution of complex packet processing tasks: FPL-3. Also, with NET-FFPF we extend the NIC-FIX architecture upwards, with packet transmission support, to create a distributed filtering platform. Experiments show NET-FFPF to be able to handle complex tasks at gigabit line-rate.

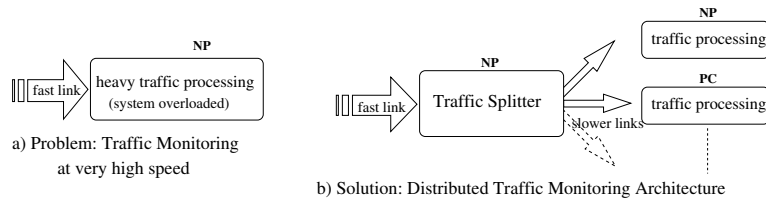


Fig. 1. Moving to distributed traffic monitoring.

This paper builds on the idea of traffic splitting that was advocated by Markatos *et al.* in [?] and Kruegel *et al.* in [?] for intrusion detection. However, we use it to provide a generic high-speed packet processing environment. Markatos *et al.* focus on packet *header* processing and automatically generate the appropriate code for the splitter (implemented on a network processor) from high-level snort rules. They show that traffic splitting improves the packet processing performance even if the splitter and all processing nodes reside on the same host. The traffic slicer in [?] employs a two-stage approach for intrusion detection where rules for traffic splitting are formed by modelling the attacks. The NET-FFPF implementation resembles the slicer in that it also mangles Ethernet frames to split the traffic. At a more fundamental level, however, NET-FFPF differs from both of the above approaches in that it allows for processing hierarchies that are arbitrarily deep and heterogeneous, whereby each level performs a part of the total computation. Moreover, NET-FFPF offers explicit support for such processing at the language level. By applying the splitter concept in a distributed fashion NET-FFPF can facilitate such diverse tasks as load balancing, traffic monitoring, firewalling and intrusion detection in a scalable manner, e.g., in enterprise gateways.

The remainder of this paper is organised as follows. In Section 2, the architecture of the distributed monitoring system and its supporting language are presented. Section 3 is devoted to the implementation details. The proposed architecture is evaluated in Section 4. Related work is discussed throughout the text and summarised in Section 5. Finally, conclusions are drawn and options for future research are presented in Section 6.

2 Architecture

2.1 High-level overview

At present, high speed network packet processing solutions need to be based on special purpose hardware such as dedicated ASIC boards or network processors (see Fig. 1a). Although faster than commodity hardware, solutions based even on these platforms are not sustainable in the long run because of a widening gap between growth rates in networking (link speed, usage patterns) and computing (cpu, main memory and bus speed).

To counter this scalability trend we propose the solution shown in Fig. 1b, which consists of splitting the incoming traffic into multiple sub-streams, and then processing these individually. Processing nodes are organised in a tree-like structure, as shown in Figure 2. By distributing these nodes over a number of possibly simple hardware devices, a flexible, scalable and cost-effective network monitoring platform can be built.

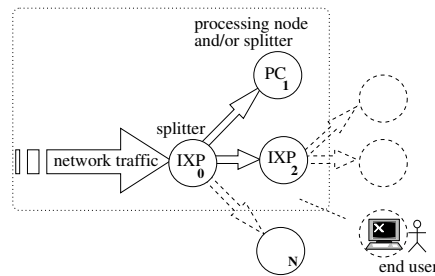


Fig. 2. Distributed system overview.

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

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

node he/she is interested in using special-purpose applications or third-party tools (e.g., tcpdump or Snort).

2.2 Distributed Abstract Processing Tree

The introduced networked processing system can be expressed in a distributed abstract processing tree (D-APT) as depicted in Figure 3. The name is derived from a close resemblance to ASTs (abstract syntax trees), as we will see later in this paper. For an easier understanding of the D-APT functionality, we use the following notations throughout the text. A D-APT is a tree composed of individual APTs, each of which has its own dedicated hardware device. An APT is built up of multiple processing elements (e.g., filters) and may be interconnected to other APTs through so-called in-nodes and out-nodes. For example, $N_{0.3}$, $N_{0.5}$ are out-nodes, while $N_{1.1}$, $N_{2.1}$ are in-nodes.

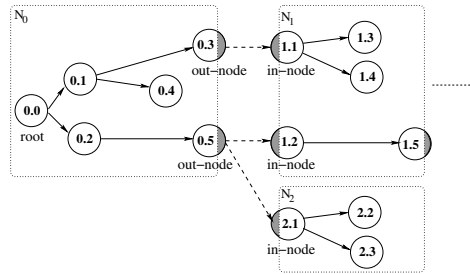


Fig. 3. Distributed Abstract Processing Tree.

By ordering the processing nodes, APTs also describe the traffic streams that flow through them. The incoming stream is decomposed into multiple substreams. Simple processing is performed at the lower levels, on the left, while more complex processing happens in the higher levels, on the right (see Fig. 3). Therefore, the amount of traffic and per packet processing ability are well balanced on each tree node.

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

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

2.3 The FPL-3 language

As our architectural design relies on explicit breakpointing, we needed to introduce this functionality into our framework. With FPL-3 we adopted a language-based approach, following our earlier experiences in this field. We designed FPL-3 specifically with these observations in mind: first, there is a need for executing tasks (e.g., payload scanning) that existing packet languages like BPF [?], Snort [?] or Windmill [?] cannot

perform. Second, special purpose devices such as network processors can be quite complex and thus are not easy to program directly. Third, we should facilitate on-demand extensions, for instance through hardware assisted functions. Finally, security issues such as user authorisation and resource constraints should be handled effectively. The previous version of the FPL-3 language, FPL-2, addressed many of these concerns. However, it lacked features fundamental to distributed processing like packet mangling and retransmission.

We will introduce the language design with an example. First, a program written for a single machine (N_0) is introduced in Figure 4. Then, the same example is ‘mapped’ onto a distributed abstract processing tree by using FPL-3 language extensions in Figure 5.

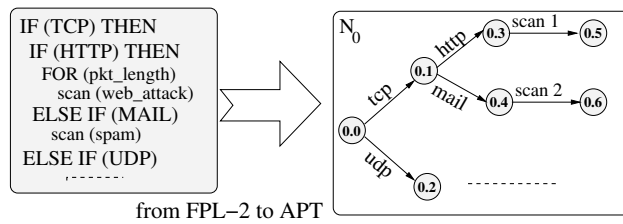


Fig. 4. Traffic monitoring program mapped onto APT.

Fig. 4 shows how the full stream is split at the root node $N_{0.0}$ into two big sub-streams: TCP and UDP. Then, the TCP sub-stream is again split into two substreams, http and mail, by the intermediate node $N_{0.1}$. Because each of these require heavy computation, they are forwarded to the leaves: $N_{0.5}$ and $N_{0.6}$, respectively.

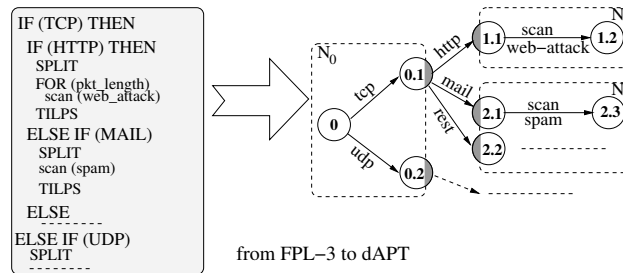


Fig. 5. mapping an APT to a D-APT using FPL-3’s SPLIT command.

When a task requires a large amount of *per-packet* processing power (e.g., a full packet scan for a worm), it becomes infeasible to perform this task on a single machine when network speeds go up. However, the layered style that all protocols reside onto a basic Ethernet frame gives enough parallelism for an eventually distributed processing environment. Thus, we give the same example, written using the SPLIT extension for a distributed processing environment and we take the hardware depicted in Figure 2 as environment. For the sake of simplicity we limit our tree to two levels. The program is mapped into the D-APT shown in Figure 5 by taking into account both the user request (our program) and a hardware description. As Figure 6 shows, the FPL-3 compiler translates the program into multiple output objects, one for each hardware device.

Currently, the object files need to be transferred and loaded by hand, but we plan to incorporate a management system into NET-FFPF that will take care of loading and releasing the runtime code as well as fetching the results of each node as one.

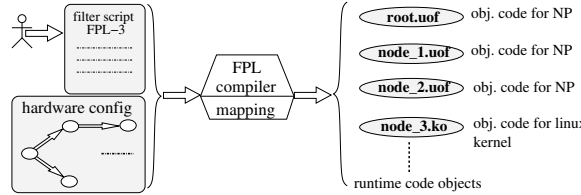


Fig. 6. User compiles an FPL-3 filter expression.

3 Implementation details

NET-FFPF builds on FFPF, a monitoring framework designed for efficient packet processing on commodity hardware, such as PCs. FFPF offers support for commonly used packet filtering tools (e.g., tcpdump, snort, libpcap) and languages (like BPF), as well as for special purpose hardware like network processors (NPs). However, it is a single-device solution. NET-FFPF extends it with distributed processing capabilities through language constructs. Currently, we have support for two target platforms: ① IXP1200 network processors and ② off-the-shelf PCs running Linux. We briefly introduce the first in the next section.

3.1 Network Processors

Network processors are designed to take advantage of the parallelism inherent in packet processing tasks by providing a number of independent stream processors (μ Engines) on a single die. The Radisys ENP 2506 network card that was used to implement NET-FFPF is displayed in Figure 7. For input, the board is equipped with two 1Gb/s 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 PCI bus ③. The IXP is built up of a single StrongARM processor running Linux and six μ Engines running no operating system whatsoever.

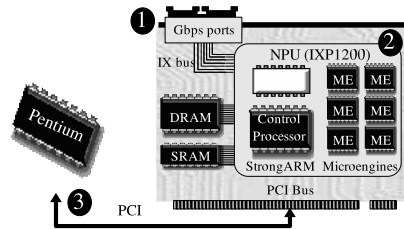


Fig. 7. Main components of the ENP-2506 board

3.2 The FPL-3 language

The FFPF programming language (FPL) was devised to give the FFPF platform a more expressive packet processing language than available in existing solutions. The latest version, FPL-2, conceptually uses a register-based virtual machine, but compiles to fully optimised object code. However, FPL-2 was designed for single node processing. We now introduce its direct descendant, FPL-3, which extends FPL-2 with constructs for distributed processing.

operator-type	operator	Data type	syntax
Arithmetic	+, -, /, *, %, --, ++	Register	R[]
Assignment	=, *=, /=, %=, +=, -= <<=, >>=, &=, ^=, =	Memory location	M[]
Logical / Relational	==, !=, >, <, >=, <=, &&, , !	Packets access:	
Bitwise	&, , ^, <<, >>	-byte . ()	PKT.B[()]
statement-type	operator	-word . ()	PKT.W[()]
if/then/else	IF (expr) THEN stmt1 ELSE stmt2 FI	-double word . ()	PKT.DW[()]
for()	FOR (initialise; test; update) stmts; BREAK; stmts; ROF	-bit in byte .	PKT.B[].U1[]
external function	INT EXTERN(filter, input, output)	-nibble in byte .	PKT.B[].U4[]
hash()	INT HASH(start_byte, len, mask)	-bit in word .	PKT.W[].U1[]
return a value	RETURN (val)	-byte in word .	PKT.W[].U8[]
transmit to	TX(table_type, table_index) or TX(table_type, id, table_index)	-bit in dword .	PKT.DW[].U1[]
split the code	SPLIT; stmts; TILPS or SPLIT(node_index); stmts; TILPS	-byte in dword .	PKT.DW[].U8[]
		-word in dword .	PKT.DW[].U16[]
		-macro	PKT.macro_name
		-ip proto	PKT.IP_PROTO
		-ip length	PKT.IP_LEN
		-etc.	customised macros

Fig. 8. FPL-3 language constructs

The FPL-3 syntax is summarised in Figure 8. It supports all common integer types and allows expressions to access any field in the packet header or payload in a friendly manner. An extensible set of macros implements a shorthand form for well-known fields, so that instead of asking for bytes nine and ten to obtain the IP header's protocol field, a user may abbreviate to 'IP_PROTO', for instance. Moreover, offsets in packets can be variable, i.e. determined by an expression. Execution safety is guaranteed by virtue of both compile-time and run-time boundary checks. As illustrated in Figure 8, most of the operators are designed in a way that resembles well-known imperative languages such as C or Pascal. We will briefly explain those constructs that are not intuitively clear.

EXTERN() construct. External functions allow users to call efficient C or hardware assisted implementations of computationally expensive functions, such as checksum calculation, or pattern matching. The concept of an 'external function' is another key to speed and system extensibility.

HASH() construct. Applies a hash function to a sequence of bytes in the packet data. This function is hardware-accelerated on IXP1200 network processors.

TX() construct. The purpose of this construct is to schedule the current packet for transmission. Currently, this operation involves overwriting the Ethernet destination address (ETH_DEST) of a packet with an entry from the MAC_table (TX_MAC). A simple use of TX() is illustrated in the example below:

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

The example shows the first TX parameter to select a table (MAC or another field, such as IP_DEST, in a future implementation) and the second parameter to be the index into the table. Note that by inserting multiple TX() calls into the same program we can easily implement packet replication and load-balancing.

SPLIT() construct. To explain SPLIT we will step through the example in Figure 5. When trying to match the given FPL-3 filter to a distributed system, the compiler detects the SPLIT construct. SPLIT tells the compiler that the code following and bounded by its TILPS construct can be split off from the main program. The example script is split into subscripts as follows: one script for the splitter node N_0 , and three more for each processing node N_1 , N_2 and N_3 , as shown in Fig. 9. Then, the scripts are compiled into object code by the native compilers.

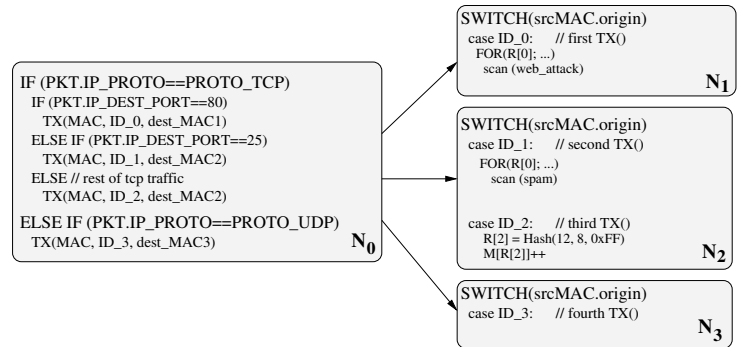


Fig. 9. SPLIT in detail.

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

The origin indicator specifies the out-node of the previous level (allowing for 64K out-points), while the 32 bit state field can be used in an application-specific way. It

allows nodes at level $N - 1$ to pass a limited amount of information to the next processing node. It is beyond the scope of this paper to discuss the use of the state field in detail. As shown in Figure 9, we can now efficiently continue the computation at level N , by using a `switch` statement on the origin indicator to jump to the appropriate point to resume. Observe that a `switch` statement is only needed if more than one out-node is connected to this in-node. Also observe that although we have not implemented this, it may be possible to generalise NET-FFPF beyond subnets by basing the splitter functionality on a tunnelled approach or by overwriting IP header fields, instead.

The compiled example program is executed as follows. The code at the root node (N_0) deals only with selective forwarding. Any packet that matches one of the IF statements has its Ethernet address fields modified and is then forwarded to the next-level nodes. The other nodes will only have to process these forwarded packets. Node N_2 for instance, receives two classes of packets forwarded from node N_0 . As illustrated in Figure 9, the classes are identified by the `origin` indicator embedded in the `ETH_SRC_ADDR` field.

Note that by passing the optional argument ‘`node_index`’ to `SPLIT` a user can force packets to be forwarded to a specific node, as in the example shown in Section 4. This can be useful when a node has special abilities well-suited to a given task, e.g., hardware-accelerated hashing.

Memory data addressing The FPL-3 language hides most of the complexities of the underlying hardware. For instance, users need not worry about reading data into a μ Engine’s registers before accessing it. Similarly, accessing bytes in memory that is not byte addressable is handled automatically by the compiler. When deemed useful, however, users may *choose* to expose some of the complexity: it is, for instance, wise to declare additional arrays in fast hardware when that is available, like in the IXP1200’s on-board SRAM. Note that NET-FFPF has no inter-node shared memory. Synchronising data across nodes is too costly relative to the high-speed packet processing task itself. To guarantee safe execution enough memory is allocated at each node to support the whole program, not just the subtask at hand. In the Ethernet implementation, limited data sharing is made possible by overwriting a now unused block in the `ETH_SRC_ADDR` field to communicate a 32 bit word *processing state* to the next hop.

3.3 The FPL-3 compiler

The FPL-3 source-to-source compiler generates straight C target code that can be further handled by any C compiler. Programs can therefore benefit from the advanced optimisers in the Intel μ Engine C compiler for IXP devices and `gcc` for commodity PCs. As a result, the object code will be heavily optimised even though we did not write an optimiser ourselves. In the near future FPL-3-compiler support will be extended to the latest Intel NP generation (e.g., IXP2400, IXP28xx).

4 Evaluation

To evaluate NET-FFPF, we execute the filter shown in Figure 10.a on various packet sizes and measure throughput (Fig. 10.b). This filter is mapped onto a distributed system

as shown in Figure 2 and the compilation results are the code objects of the three sub-filters highlighted in same picture.

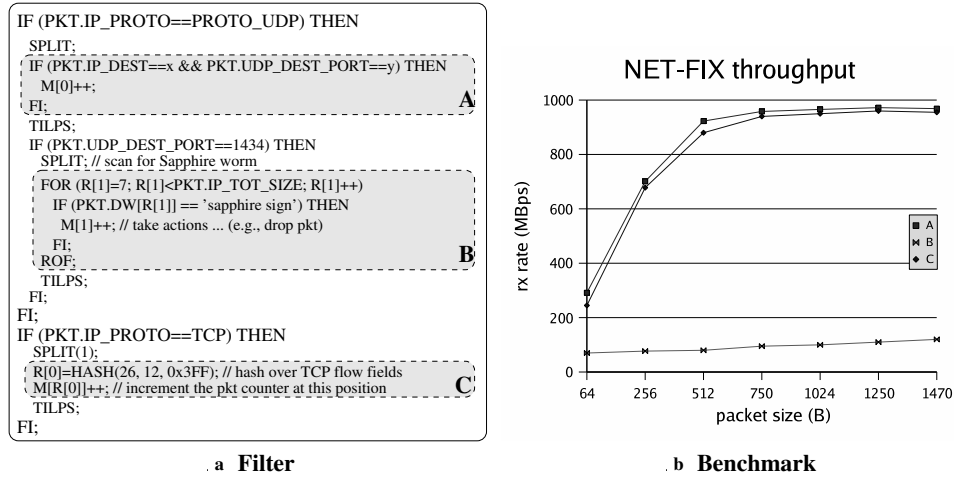


Fig. 10. Filtering results

We note that only *A* is a traditional per-packet counter. The other two gather per-flow information. As the hash function used in *C* utilises dedicated hardware support, we push (by providing the parameter `node_index`) the *C* filter onto the same hardware node as *A* filter – N_1 . In *B*, a loop iterates over the whole UDP packet payload. While the filters *A* and *C* are easily performed on an NP even at high speed (1Gb/s), the *B* filter incurs so much processing that even an IXP1200 network processor cannot handle its stream at such speed. As the processing results show, using this specific hardware, we can process the full packet data up to only ca. 100Mb/s. Therefore, we let the filter split to the second node – N_2 and we can successfully process all the packets for a particular UDP port (assuming the packets related to a specific worm are within 100Mb/s bandwidth). If more bandwidth needed, then more processing nodes have to be involved.

As illustrated in Fig. 10, just as for the *B* filter, throughput also drops for the simple filters *A* and *C* when processing smaller packets. However, these drops occur for a different reason, namely because the *receiving* μ Engine simply cannot keep up.

Demonstrating the simplicity a concise special purpose language like FPL-3 brings, a naive load balancing program for a web-server is shown below. A hash over the flow fields of each incoming packet determines to which node the packet is forwarded. In a handful lines of code the web traffic is split into three equal substreams.

```

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

```

5 Related work

Using traffic splitters for increased packet processing efficiency on a single host was previously explored in [?] and [?]. Our architecture differs in that it supports distributed and heterogeneous multi-level processing. In addition, we provide explicit language support for this purpose. As shown in [?], it is efficient to use a source-to-source compiler from a generic language (Snort Intrusion Detection System) to a back-end language supported by the targeted hardware compiler (Intel μ EngineC). We propose a more flexible and easy to use language as front-end for users. Moreover, our FPL-3 language is designed and implemented for heterogeneous targets in a distributed multi-level system.

Many tools for monitoring are based on BPF in the kernel [?]. Filtering and processing in network cards is also promoted by some Juniper routers [?]. However, they lack features introduced in NET-FFPF such as extended language constructs, in-place packet handling and a distributed processing environment. BPF+ [?] shows how an intermediate representation of BPF can be optimised, and how just-in-time-compilation can be used to produce efficient native filtering code. FPL-3 relies on `gcc`'s optimisation techniques and on external functions for expensive operations.

Like FPL-3 and DPF, Windmill protocol filters also target high-performance by compiling filters into native code [?]. And like MPF, Windmill explicitly supports multiple applications with overlapping filters. However, compared to FPL-3, Windmill filters are fairly simple conjunctions of header field predicates.

Nprobe is aimed at monitoring multiple protocols [?] and is therefore, like Windmill, geared towards spanning protocol stacks. Also, Nprobe focuses on disk bandwidth limitations and for this reason captures as few bytes of the packets as possible. NET-FFPF has *no priori* notion of protocol stacks and supports payload processing.

The SCAMPI architecture also pushes processing to the NIC [?]. It assumes that hardware can write packets immediately into host memory (e.g., by using DAG cards [?]) and implements access to packet buffers through a userspace daemon. SCAMPI does not support user-provided external functions, powerful languages such as FPL-3 or complex filtergraphs.

Related to the distributed architecture of NET-FFPF are the Lobster EU project and Netbait Distributed Service [?] that aim at European-scale passive monitoring and at planetary-scale worm detection, respectively. Netbait, for instance, targets high-level processing using commodity hardware. Therefore, these initiatives could benefit from using the FPL-3 language and its NET-FFPF execution environment as low-level layer.

6 Conclusions and future work

This paper presented the NET-FFPF distributed network processing environment and its FPL-3 programming language, which enable users to process network traffic at high speeds by distributing tasks over a network of commodity and/or special purpose devices such as PCs and network processors. A task is distributed by constructing a processing tree that executes simple tasks such as splitting traffic near the root of the tree while executing more demanding tasks at the lesser-travelled leaves. Explicit language

support in FPL-3 enables us to efficiently map a program to such a tree. The experimental results show that NET-FFPF can outperform traditional packet filters by processing at Gbps line rate even on a small-scale (two node) testbed.

In the future, we plan to extend NET-FFPF with a management environment that can take care of object code loading and program instantiation. A first version of this management subsystem will act only when a user issues a recompile request. We envision a later version to be able to automatically respond to changes in its environment like the increase of specific traffic (e.g., tcp because of a malicious worm) or availability of new hardware in the system (e.g., a system upgrade). We also plan to have the FPL-3 compiler optimise code placement. As a result, tasks that are known to be CPU intensive – such as packet inspection, hashing or CRC generation – will be automatically sent to optimal target machines, for instance those with hardware assisted hash functions (NPs).

Acknowledgements

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

References