Journal of
Sensor and Actuator Networks

Article

Enif-Lang: A Specialized Language for Programming Network Functions on Commodity Hardware

Nicola Bonelli, Stefano Giordano and Gregorio Procissi *

Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Via G. Caruso 16, 56122 Pisa, Italy; nicola.bonelli@for.unipi.it (N.B.); stefano.giordano@unipi.it (S.G.)
* Correspondence: gregorio.procissi@unipi.it

Received: 30 June 2018; Accepted: 3 August 2018; Published: 7 August 2018

Abstract: The maturity level reached by today's commodity platforms makes even low-cost PCs viable alternatives to dedicated hardware to implement real network functions without sacrificing performance. Indeed, the availability of multi-core processing packages and multi-queue network interfaces that can be managed by accelerated I/O frameworks, provides off-the-shelf servers with the necessary power capability for running a broad variety of network applications with near hardware-class performance. At the same time, the introduction of the Software Defined Networks (SDN) and the Network Functions Virtualization (NFV) paradigms call for new programming abstractions and tools to allow this new class of network devices to be flexibly configured and functionally repurposed from the network control plane. The paper presents the ongoing work towards Enif-Lang (Enhanced Network processIng Functional Language), a functional language for programming network functions over generic middleboxes running the Linux operating system. The language addresses concurrent programming by design and is targeted at developing simple stand-alone applications as well as pre-processing stages of packet elaborations. Enif-Lang is implemented as a Domain Specific Language embedded in the Haskell language and inherits the main principles of its ancestor, including the strong typedness and the concept of function compositions. Complex network functions are implemented by composing a set of elementary operations (primitives) by means of a compact yet expressive language grammar. Throughout the paper, the description of the design principles and features of Enif-Lang are accompanied by examples and use cases. In addition, a preliminary performance assessment is carried out to prove the effectiveness of the language for developing practical applications with the performance level required by 5G systems and the Tactile Internet.

Keywords: commodity hardware; concurrent programming; network function programmability; functional composition; stateful processing

1. Introduction

The success of the Software Defined Networks (SDN) and the Network Function Virtualization (NFV) paradigms has naturally fueled the model of *softwarized networks* in which specialized hardware is replaced by general-purpose platforms and network functions are implemented as software applications instantiated through the network control plane. Both SDN and NFV propose a dramatic change in the way network operations and services are conceived and push *programmability* and *reconfigurability* as crucial keywords of a new generation of network devices.

The adoption of commodity hardware as the new "metal" for network equipments is well justified by its consolidated evolution towards powerful multi-core platforms that commonly accommodate a new family of flexible multi-gigabit network interfaces. At the same time, the attained maturity of commodity hardware has been accompanied by significant progresses of the software capabilities in handling even high-speed network traffic. Indeed, a flourish literature about effective I/O operations has lately appeared in the scientific arena by proposing accelerated I/O engines for 10+ Gbps line rates such as PF_RING [1], PF_RING ZC (Zero Copy) [2], Netmap [3], Data Plane Development Kit (DPDK) [4] and Packet Family Queue (PFQ) [5].

In this scenario, the adoption of commodity PCs as both traditional network nodes and middleboxes becomes an almost natural consequence. This is perfectly in-line with the evolving model of SDN that, from the pragmatic approach of OpenFlow [6] in which network nodes are mere enforcer of the decisions taken by the network brain (the controller), is slowly migrating towards a model in which a portion of the logic is offloaded to the network devices. As a result, device programmability is becoming a very hot topic in research and new data-plane programming models are emerging. Indeed, starting from the basic *stateless* Match-Action paradigm proposed by OpenFlow, several other approaches (e.g., [7–9]) have been proposed to enable *stateful* processing within network devices. However, stateful operations do not straightforwardly cope with parallel processing when performance matters. The most common issue is represented by state sharing among the flows processed by different cores. Such a condition becomes even more critical when multiple applications run concurrently on the same set of network interfaces.

This paper presents Enif-Lang (Enhanced Network processIng Functional Language), a simple, compact, expressive and robust programming language specifically tailored to network traffic processing for multi-core PCs running Linux Operating System (OS).

Originally born as PFQ-Lang [5,10], the language has lately evolved quite a lot and is still a topic of ongoing research. The seminal version of the language [11] was designed to let applications run in the *kernel-space* of the Linux OS within PFQ. The new version, instead, enables *user-space* usage and it is entirely implemented as a Domain Specific Language (DSL) within Haskell. As such, its basic behavior is now independent of the underlying packet I/O framework, although some of its advanced features (mainly in the transmission side) need to be supported by the I/O system underneath. In addition, the current version now integrates a *runtime* platform that orchestrates the language operations and addresses the management of the state to allow core isolation in concurrent programming scenario. Furthermore, with respect to [11], the current document presents in more details the language features, including a simple application that is used to exemplify the language usage as well as a benchmark to give evidence of the performance that can be achieved.

The implementation maturity of Enif-Lang is still at the experimental level. The language description refers to the current version of the prototype, hence specific grammar details may be subject to change while the research progresses.

The rest of the paper is organized as follows. Section 2 introduces the technical background and related work in the area of network programming with commodity hardware. Section 3 gives a high level overview of Enif-Lang by introducing the main feature and its application scope. Section 4 delves into the details of the language by presenting the specific classes of instructions (primitives) that are combined together into Enif-Lang programs, as described next in Section 5. Section 6 addresses the management of stateful operations in multi-applications context and details the *clustering* approach adopted by Enif-Lang to guarantee core isolation for performance reasons. Section 7 presents a simple "Hello word" monitoring application written in Enif-Lang and performing flow tracking over live data. The example is used to show the expressiveness of the language itself as well as to assess the impact of the language machinery on the overall performance. Section 8 extends the scope of usage of Enif-Lang by presenting two practical examples of stateless and stateful network functions implementing a *network load-balancer* and a *port-knocking firewall*. Finally, Section 9 draws conclusions and final remarks.

2. Background

Network softwarization requires on one hand powerful computing platforms, and, on the other hand, effective software tools to handle and process high rates of traffic. If the technological maturity reached by off-the-shelf hardware platforms does make general-purpose servers viable candidate for implementing real network nodes, a significant effort has lately been directed towards the software accelerated approaches for efficient packet handling. At the capture level, much research has focused on the main limitations of general-purpose operating systems and proposed different solutions that bypass either the OS network stack or the entire operating system. A thorough review of such approaches is contained in the papers [12–14] along with their comparison and usage guidelines.

From a chronological point of view, PF_RING [1] was one of the first accelerated engines for 1 Gbps links. More recently, PF_RING ZC (Zero Copy) [2] and Netmap [3] allow a single CPU to retrieve short sized packets up to a full 10 Gbps line rate by memory mapping the ring descriptors of Network Interface Cards (NICs) at the user space. Intel's DPDK [4] is a very popular engine that accelerate packet capture by bypassing the operating system. In addition, DPDK provides a user-space framework for packet processing on multi-core architectures running Linux. PFQ [5] is a software acceleration engine built upon standard network device drivers that primarily focuses on programmable packet fan-out.

Other proposals include OpenOnLoad [15] that rebuilds the network stack for SolarFlare products to seamlessly accelerate existing applications and HPCAP [16], a packet capture engine that focuses on the efficient storage of live traffic into non-volatile devices and to perform timestamping and delivery to multiple listeners at user-space. NetSlices [17] provides operating system abstractions towards hardware resources, such as multi-core processors and multi-queue network adapters.

Recently, the `pcap` library has been extended [18] to support the fan-out mechanism over different capture engines, thus allowing `pcap` based network applications to select the packet distribution flavor straight from the `libpcap` Application Programming Interface (API) without the need for managing raw socket details and hardware configurations. Finally, it is worth mentioning that the Linux kernel itself has significantly improved its capture performance with the adoption of the TPACKET (version 3) socket that integrates PACKET_MMAP [19] for efficient packet memory mapping towards the user-space.

At a higher level, software acceleration has been integrated in soft-based switches and routers, seldom taking advantage of the previously mentioned frameworks. Packetshader [20] was one of the first proposals of a high performance software routing based on a heavily modified driver and on GPU acceleration. A thorough investigation about the design and implementation of high performance software routers by distributing workload across cores was also presented in Ref. [21]. The Click [22] modular router has determined a significant number of follow-up works. Routebricks [23] proposes routing clusters to improve the performance of software-based routing. In Ref. [24], the Netmap I/O framework has been used to accelerate Click itself while FastClick [25] features I/O batching and advanced multi-processing techniques and integrates both Netmap and DPDK. Snap [26] takes advantage of offloading computation intensive processes to GPUs in order to improve the Click performance. PFQ proved to effectively accelerate the OpenFlow software switch OfSoftSwitch [27] and the monitoring framework Blockmon [28] on commodity PCs. Finally, the kernel bypass mode of Ethernet I/O and the Lua scripting language are used in the Snabb switch [29] to build a fast and easy to use networking toolkit.

All the above approaches prove that even high-speed packet handling on general purpose platforms is no longer an issue. However, when it comes to *programmability*, the very few specialized network programming models that have emerged so far mostly originate from the switching domain. Beyond OpenFlow, the most successful proposal is currently P4 [7], a high-level language that allows to define the data-plane processing operations for programmable switches. However, the programming abstractions inherited by the switching domain typically show clear limitations when the processing scope become more complex and include *stateful operations*. More sophisticated approaches like OpenState [8] and Open Packet Processor (OPP) [30] specifically address the management of stateful processing on hardware platforms through the use of eXtended Finite State Machines (XFSM).

If the above proposals mainly focus on hardware devices, not much has been done in terms of specialized network programming languages for general-purpose platforms, and most of the programming effort is still addressed by traditional general-purpose languages such as C, C++, Lua, etc. Indeed, the most recent proposals are limited to the imperative language Pyretic [31] (from the Frenetic [32] family of network programming languages) for the description of the data plane logic of an SDN network, and to Streamline [33] for configuring I/O paths to applications through the operating system. In this area, the Vector Programming Processing (VPP) [34] is also worth being mentioned as it originates from the Cisco world for switch/router functionalities. Finally, embedded Berkley Packet Filter eBPF [35] is recently emerging as an in-kernel advanced programming approach for packet processing and data plane programming.

So, why Enif-Lang?

Enif-Lang enters the arena of the above listed approaches by providing a simple tool for programming generic *network functions*. As such, this is not specifically targeted at any specific application domain (as opposed to OpenFlow, P4, etc. that focus on switching), while being designed to help a network programmer by means of a compact yet expressive language grammar and a rich set of primitives. In fact, the set of typical applications and Virtual Network Functions (VNFs) that can be either fully developed in Enif-Lang or accelerated through a first stage of Enif-Langwritten pre-processing application includes soft switches, network filters, traffic shapers, load balancers, network intrusion detection systems (NIDS), deep packet inspection (DPI) modules, etc.

Enif-Lang targets by design programming safety through the strong typedeness feature inherited from the underlying functional philosophy. This, in turn, guarantees no undefined application behavior as well as nearly certain correctness upon successful code compiling.

Finally, the language is conceived to guarantee high-level application speed. Indeed, the implementation of network functions, especially in 5G and Tactile Internet scenarios, cannot leave performance aside of their basic requirements, and the adopted programming abstraction cannot make an exception. Enif-Lang framework proves to obey the performance requirements (up to multi-gigabit line-rate speed) by taking advantage of the efficiency of the underlying Haskell language as well as by explicitly enabling *parallel processing* through a proper management of stateful operations.

3. Enif-Lang at a Glance

Enif-Lang is a functional language designed to develop applications that run in the user-space of Linux systems. The language is implemented as a declarative Domain Specific Language (DSL) on top of the Haskell Language from which it borrows similar constructs and semantic.

At the very beginning, Enif-Lang was designed as an extension of PFQ-Lang [5]. Not only the first version [11] was based on a similar grammar, but it was also built on top of a collection of composable functions implemented in GNU C language (much like PFQ–Lang). The experimental compiler was meant to enforce the correctness of the grammar and to translate the source code into a *hybrid binary-based* representation suitable to be serialized in kernel space and executed by a runtime, implemented as a dedicated module.

After that, we decided to move the implementation of the language to the user-space for a number of reasons, including the simplicity of development, the portability (Enif-Lang is theoretically no longer Linux dependent), the possibility to reuse other tool chains (the GHC – Glasgow Haskell Compiler) and, more importantly, to ease the interoperability with other applications and languages.

The net result is no longer a DSL for the abstract syntax tree only, but rather an *embedded* DSL (eDSL) in which functions can either be implemented by means of Enif-Lang itself (possibly using external modules and libraries), or by means of other languages (such as, Haskell and C) via the Foreign Functions Interface (FFI). As a matter of fact, the new language turns out to be more flexible and extensible without compromising performance.

The design principles of Enif-Lang are those typical of functional languages and include strong static type safety (i.e., no undefined behavior is allowed) and data (i.e., packets) immutability. An underlying *runtime* framework handles by design data contention and parallelism, implements packets duplication (copy-on-write) to allow packet mutability and to better achieve parallelism by avoiding head-of-line blocking.

Figure 1 shows the full processing model. Enif-Lang programs are compiled through the standard Glasgow Haskell Compiler (GHC) as `.so` *shared objects* which are loaded by the runtime and attached to specific hooks. Such hooks can be attached to physical network ports (or possibly channels), to `pcap` files and so forth.

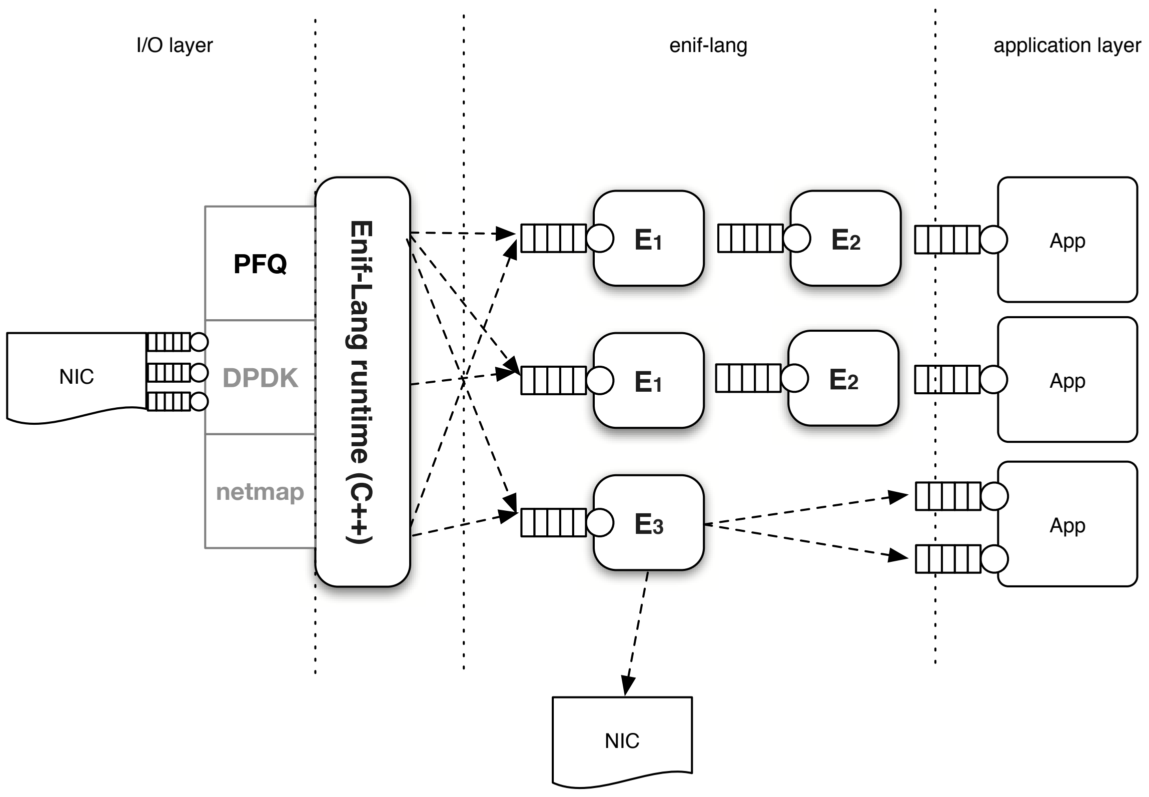


Figure 1. The Enif-Lang abstract processing model.

The runtime is responsible to handle traffic from the network, to hide the grammar and the semantic differences of the various underlying sockets (e.g., PFQ, DPDK, TPACKET3, etc.) by means of abstract engines, and to expose a set of hooks where the user can attach Enif-Lang programs. In addition, it allows multiple applications to consume packets from the same network devices.

Packets retrieved by the runtime at the physical network interfaces are steered towards the Enif-Lang apps (the E_i blocks in the picture) by maintaining per-flow consistency. At this stage, computations are executed according to the description provided by Enif-Lang programs before being finally delivered to the selected *endpoints*, i.e., to network cards, application threads, the OS kernel, etc. Notice that the last layer of applications is optional: indeed, Enif-Lang is expressive enough to build simple stand-alone network applications. However, the most common use of Enif-Lang is to build pre-processing stages of elaborations (e.g., filtering, marking, etc.) while delegating more refined elaborations to traditional applications such as IDSs, visual analytics, etc., that receive data through `tun` interfaces.

The entry point of Enif-Lang program is the function `enif-main`. To amortize the cost of invoking the `enif-main` function, a batching technique is also implemented in the runtime. Therefore, the most recent implementation of the Enif-Lang offers an entry-point for the main function which takes a queue of packets instead of just a single one. In the current version, packets cannot be re-injected in the processing loop, but rather can be distributed to virtual or real interfaces, to enable the implementation of VNFs entirely written in Enif-Lang.

Enif-Lang programs are formally expressed as the composition of pure and effective functions, called *actions*, that perform network operations on top of packets. Actions are formally modelled around the concept of *monad* [10] that provides the theoretical support for composing effective functions (such as those performing I/O, those that handle the state associated with a packet or a flow, and so on) while maintaining the pure functionality of the language. As it will be elaborated upon in the following sections, Enif-Lang instructions include most of the common packet processing primitives for packet forwarding, filtering, steering, logging and statistics retrieval.

4. Enif-Lang Primitives

Enif-Lang provides a rather expressive set of built-in primitives as well as a complete library of functions to describe generic packet processing. Depending on their purposes, such functions are classified into different common categories, as detailed in the following.

Meters and Counters

Meters and counters are local and global generic registers used to temporarily store the results of computations. They are mainly derived from the Haskell language and they can be assessed through specialized functions such as the `modifyCounter` primitive of the example below:

```
udpCounter = counter 0
enif_main :: Packet -> ActionIO ()
enif_main pkt =
when (is_udp pkt) $ modifyCounter udpCounter (+1)
```

Counters offer a mean to handle global states that can either be used directly from the language or can be exported by the runtime to inspect the behavior of an instrumented Enif-Lang application.

Predicates

Predicates are pure functions that take an arbitrary number of arguments (possibly none) plus the current packet and return a boolean value. Such functions are either used within the `if-then-else` statement, or passed as argument to high-order-functions to specialize their behavior.

The language includes a set of primitive predicates that compose together to implement more complex ones. The default library includes predicates for the most common protocols. As a convention, the names of such functions are prefixed by `is_` or `has_`, when meaningful. Examples are `is_tcp, is_udp, is_icmp, has_addr CIDR, is_rtp, is_sip, is_gtp` or `has_port 80, has_addr ‘‘192.168.0.0/24’’`.

The following snippets show the use of predicates within both the `if-then-else` and the `when` statements:

```
if (not is_tcp pkt)
then pass pkt
else drop pkt

when (has_addr "192.168.0.0/24" pkt) $ do
forward pkt "eth1"
```

Notice that the last piece of code includes the use of the `do` notation, which is mandatory to handle multiple statements. This indeed is a central element of Enif-Lang and will be further elaborated in Section 5.

Combinators

Enif-Lang provides a set of combinators, namely functions designed to combine predicates together. In particular, the composition of predicates is enabled by the logical `or`, `and`, `xor` and `not` functions, also available in the form of operators: ||, `&&`, ^^. In addition, the language offers special non-boolean functions to force filter inversion (`inv`) and parallelization (`par`) with the following signature:

```
inv :: (Packet -> ActionIO Packet) -> (Packet -> ActionIO Packet)
par :: (Packet -> ActionIO Packet) -> (Packet -> ActionIO Packet)
                                   -> (Packet -> ActionIO Packet)
```

As an example, in the following line of code, the combinator `par` is used to parallelize the `udp` and `tcp` filters before delivering packets to the OS kernel:

```
par tcp udp >=> kernel
```

Properties

Properties are functions that take an arbitrary number of arguments and a packet as input and return a value associated with the packet. Typical examples are hash functions computed over a portion of a packet, header field extractors, state retrievals, etc. The Enif-Lang library includes a large number of properties for the most common protocols (IP, TCP, UDP, ICMP, etc.), as well as a set of field value extractors from arbitrary protocols (upon specifying the offset and the size of the field). Examples of properties are `ip_tos, ip_tot_len, ip_id, ip_frag, ip_ttl, get_mark, get_state, tcp_source, tcp_dest`, and so on. In the following piece of code, the value of TTL is first extracted from the IP header of packets while the state value is later accessed and compared to 10 before packet delivery to the OS kernel:

```
filter (ip_ttl pkt < 5) pkt
when (get_state pkt == 10) kernel
```

Comparators

Comparators are among the most commonly used functions in any languages to perform a comparison between a property and a specified value. Along with the standard operators <, <=, >, >=, == and /=, the Enif-Lang library provides some network-specific functions, such as `any_bit` and `all_bit` to check whether some (or all) bits of a given mask are set. The two functions have identical signature:

```
any_bit :: (Packet -> Word) -> Word -> Packet -> Bool
all_bit :: (Packet -> Word) -> Word -> Packet -> Bool
```

and the following example shows the usage this of the `any_bit` comparator within the `if` statement to check the flags of TCP packets:

```
if (any_bit tcp_flags (SYN|ACK) pkt)
```

Filters

Filters are effective functions that break the chain of processing when the packet does not match a given condition. The Enif-Lang library is equipped with a wide range of filters for the most common protocols. In short, a filter is a function whose output action can be either *Pass* or *Drop*. Examples of common filters are `ip`, `tcp`, `udp`, `port`, `src_port`, `dst_port`, `addr`, `src_addr`, `dst_addr`, etc. As a further example, the more generic `l3_proto` and `l4_proto` have the following signature:

```
l3_proto, l4_proto :: Int16 -> Packet -> ActionIO Packet
```

In the following snippet, the `l3_proto` function is used to recognize packets carrying the *wake-on-line* protocol; a log message is next prepared upon the composition to the `log_msg` function:

```
l3_proto 0x842 >=> log_msg ‘‘Wake-on-LAN!’’
```

Monadic Functions

Monadic functions are the typical primitives of functional languages as they allow for enriching the output with extra information while still obeying the functional principle. In Enif-Lang, they take an arbitrary number of arguments together with a packet, and produce a packet with an associated action. The available actions are: *Pass*, *Drop* (used by filters), *Broadcast*, *Dispatch*, *Steer* and *DoubleSteer* (used by steering functions).

Common monadic functions are `when` and `unless`, and are used to implement conditional statements. Special types of monadic functions are those in charge of distributing packets to the final end-points. Such functions, named *steering functions*, provide fine-grained control of packet dispatching across end-points. Their generic signature is given by:

```
Steering :: Arg1 -> Arg2 ... -> Packet -> ActionIO Packet
```

and a few examples are `steer_flow`, `steer_p2p`, `steer_link`, etc., used to balance the traffic among multiple endpoints with different flow consistency guarantees.

High-Order Functions

High-order functions takes either functions or effective functions as argument. Examples of such functions are the instructions `conditional, when, unless, inv, par, filter`, etc. In addition, this category also includes the built in `if-then-else` construct, whose support is borrowed from the underlying Haskell language.

5. Enif-Lang Programs

As previously mentioned, Enif-Lang is built around the concepts of pure and effective functions (actions). Actions act as the processing primitives and are mainly used for typical network operations **such as** packet forwarding, filtering, steering, logging and sharing state across functions.

Pure and monadic functions are then composed together by means of either the `do` notation (slightly different from that of Haskell), or through the Kleisli operator >=>, and follow the precise rules of composition set forth in [10].

The following simple program exemplifies the use of the `do` notation to (i) select DNS packets only, (ii) log the results and (iii) forward them to the `eth1` network interface:

```
dns :: Packet -> ActionIO Packet
dns pkt = port 53 pkt

enif_main :: Packet -> ActionIO ()
enif_main pkt = do
                dns pkt
                log_packet pkt
                forward "eth1"
```

The snippet below, instead, shows the use of the Haskell pointfree style and the Kleisli operator to provide an alternative description of the same operations:

```
enif_main = dns >=> log_packet >=> forward "eth1".
```

In general, the previous line of code gives a good example of the expressiveness of Enif-Lang, which makes it possible to describe even complex pipelines through a very concise grammar.

The following example describes a *stateless* filter that allows TCP packets to be logged and passed back to the kernel of the operating system (notice that forwarding packets to the OS kernel and to network interfaces requires the underlying I/O framework to support a feedback channel):

```
enif_main = tcp >=> log >=> kernel.
```

The natural and easy step ahead is represented by programs in which the elaboration requires the use of a *per-computation state*:

```
is_http = has_dst_port 80,

pass_to_kernel pkt =
    state <- get_state
    when (state == http_traffic)
         kernel pkt,

mirror_to_port pkt port =
    state <- get_state
    when (state == other_traffic)
         (forward port pkt),

process pkt = if (not is_tcp pkt)
            then drop
            else do pass_to_kernel pkt
                    mirror_to_port pkt "eth2"),
http_traffic = 1,
other_traffic = 2,

enif_main :: Packet -> ActionIO ()
enif_main pkt =  do
  if (is_http pkt)
    then put_state http_traffic
    else put_state other_traffic
  process.
```

In the example, functions like `put_state`, `get_state` are implemented as properties and act as an implicit extra parameter for all of the functions. The overall computation takes advantage of a per-packet state, which, however, expires at the end of the packet processing itself. Hence, this specific programming model cannot be applied to implement elaborations that require the storage of stateful information across different packets. In all such cases, the state must be kept persistent (and consistent) over *entire flows of packets*, as it will be shown in detail in the next section.

6. State Management

One of the key issues addressed by Enif-Lang is the efficient handling of computation resources to allow concurrent programming. Generally speaking, network applications find great benefits from parallel computing as long as the processing cores can work in perfect isolation without sharing any kind of information with other cores. Indeed, data sharing among cores (especially at high rates) is not recommended due to synchronization issues (to avoid race conditions), cache (in)coherence among cores, and because of the mandatory use of Compare-and-Swap (CAS) and Software Transactional Memory (STM) techniques that have a dramatic impact on performance.

As a general rule, core isolation is typically obtained by selecting a proper *flow description* and by distributing traffic according to such a per-flow policy. Indeed, this is the operational behavior of the runtime framework that delivers the packets from the underlying I/O engine to the different chains of processing by maintaining per-flow consistency. Hence, the key point is to provide a suitable definition of packet flow. Packet flows are defined through flow-keys (such as, IP addresses, canonical 5-tuples, etc.). In the Enif-Lang context, a generic flow-key consists of the concatenation of an arbitrary number of packet header fields. For example, the next two definitions of flow are associated with the classical IP 5-tuple and the pair source/destination IP addresses (*mega-flow*), respectively:

```
IP_SRC | IP_DST | PORT_SRC | PORT_DST | PROTOCOL
IP_SRC | IP_DST
```

As shown in Figure 2, an application running three threads (or, equivalently, three instances of a process) can be parallelized by simply distributing traffic to the threads (processes) on the basis of a hash function computed over the key represented by the flow descriptor.

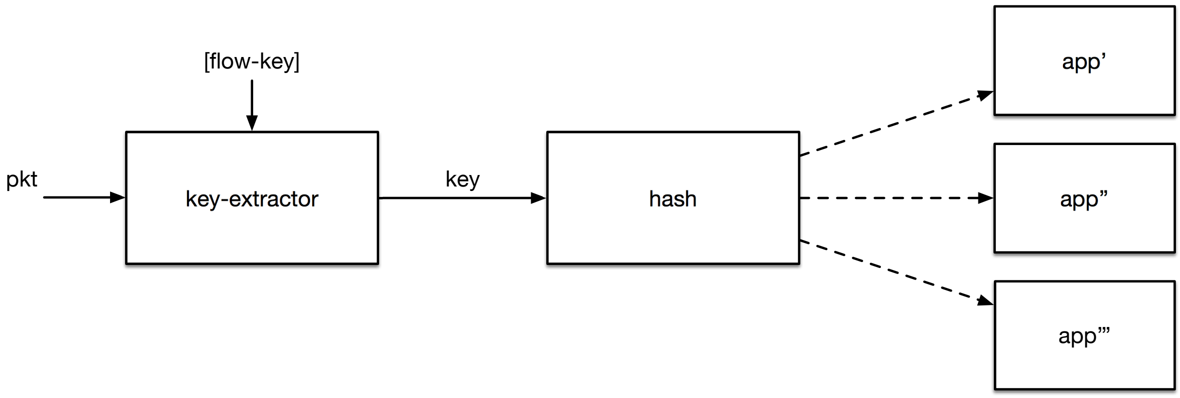


Figure 2. Packet distribution to multiple threads of an application.

6.1. Multiple Applications

The computational power of todays commodity servers does allow multiple applications to run at the same time on the same cores. However, different applications may take care of very different operations and the definition of flows may be significantly different. In all such cases, traffic distribution still provides the key for enabling parallelism as long as the concept of flow key is replaced by the the concept of *largest common flow key (LCFK)*. Indeed, each application specifies its own flow key. The LCKF is obtained by the runtime through the *bitwise intersection* of all such keys, and it is used as the new key for performing hash-based packet distribution.

Figure 3 shows three instances of two applications running on three cores. The flow key specified by application 1 is represented by the canonical IP 5-tuple while the second application specifies the triple <*source IP, dest. IP, protocol*> as its flow descriptor. The intersection of the two flow keys is therefore given by the latter and passed to the key-extractor module to pull out the hash key. Notice that the flow key of app1 is more specific than that of app2. As such, all of the packets that match the flow key of app1 will also match the one of app2.

Once traffic is split, the current abstraction guarantees that packets of the same flow are processed in the Enif-Lang stage sequentially on the same core. This automatically ensures the flow consistency for all the packets and their related states and prevents from inter-core data-sharing. It is worth noticing that all instances of applications 1 and 2 are executed sequentially on each core and receive the same input traffic distributed according to the LCFK (visually depicted by the dashed arrow that originates on the hash module). In addition, the order of execution of the apps is not relevant thanks to the packet immutability property enforced by the functional programming model.

```
• App1 flow key: 5-tuple          [ IP_SRC | IP_DST | PORT_SRC | PORT_DST | PROTO ]

• App2 flow key: 3-tuple          [ IP_SRC | IP_DST |    -     |    -     | PROTO ]

  => Largest Common Flow Key      [ IP_SRC | IP_DST |    -     |    -     | PROTO ]
```

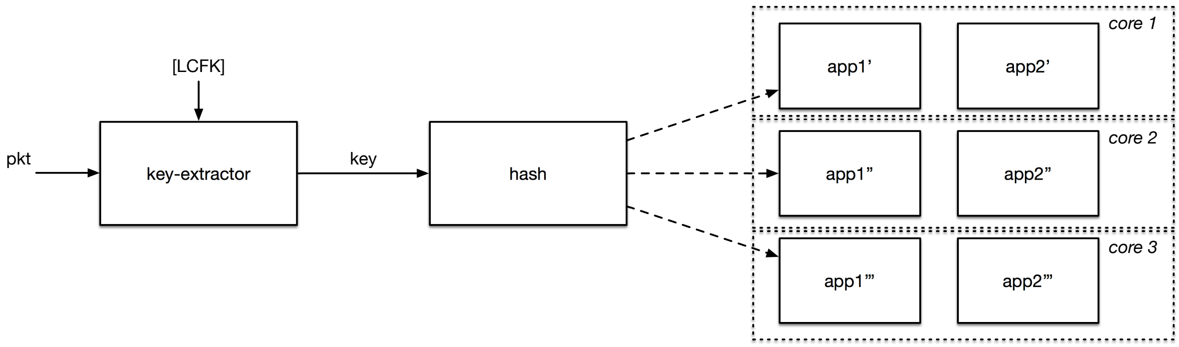


Figure 3. LCFK for packet distribution to multiple applications.

6.2. Clustering

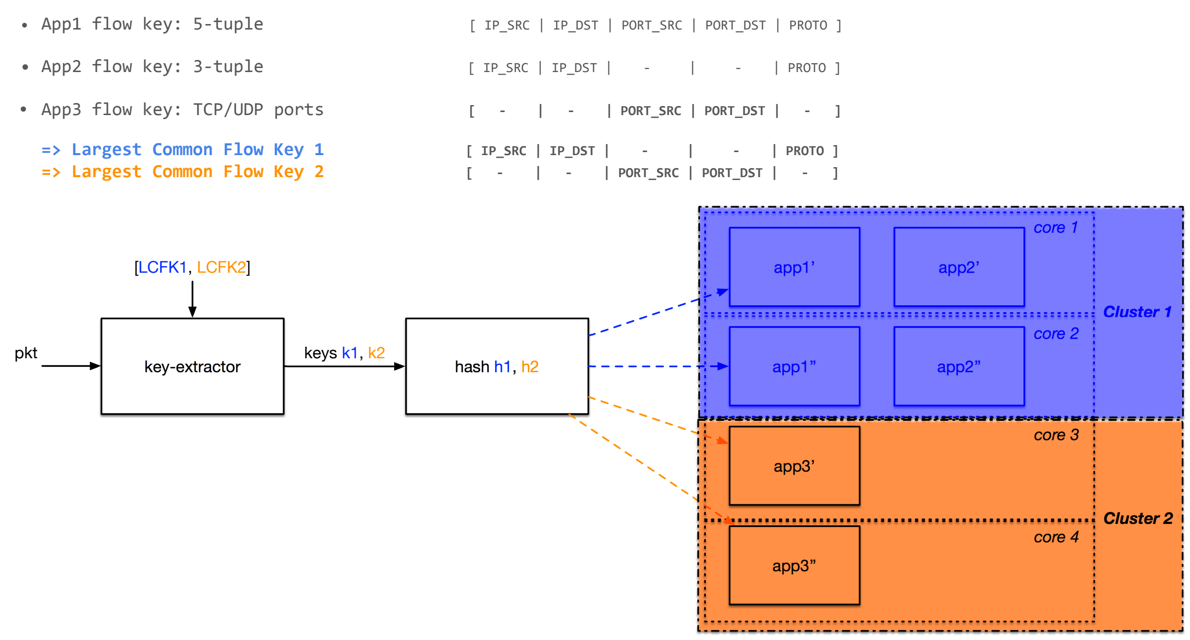
Unfortunately, it is not always possible to find a proper flow definition that guarantees data to be strictly confined within separated cores. As an example, consider the problem of parallelizing the MAC self-learning algorithm of Ethernet switch. For each Ethernet frame reaching the switch, the incoming physical port along with its MAC source address are stored in a forwarding table that gets continuously populated so as to allow the device to discover the L2 addresses of the connected hosts. The acquired information is then used by the switch to select the proper output interface as a result of MAC destination address lookup. However, it should be clear that in this case there is no definition of flow key that guarantees core isolation, even within the same application. Indeed, if packets are distributed by hashing on the MAC *destination* address, they will likely not find the corresponding entry in the forwarding table, as the latter is filled on the basis of the MAC *source* address. In other words, the forwarding table must be shared upon all of the cores operating the self-learning algorithm.

A more general case is represented by the following example, in which three applications define the following three flow keys:

```
App1  flow key: IP 5-tuple      [ IP_SRC | IP_DST | PORT_SRC | PORT_DST | PROTO ]
App2: flow key: IP 3-tuple      [ IP_SRC | IP_DST |    -     |    -     | PROTO ]
App3: flow key: TCP/UDP ports   [   -    |   -    | PORT_SRC | PORT_DST |   -   ]
=> LCFK:                        [   -    |   -    |    -     |    -     |   -   ]
```

for which the LCFK is empty.

As shown in Figure 4, this special case can be conveniently handled by partitioning the applications into *clusters* sharing a common sub-key and by introducing shallow copies of packets to feed each of them. In practice, all amounts of traffic are replicated on both cluster 1 (accommodating apps 1 and 2) and cluster 2. Within the same cluster, traffic is load balanced across the participating threads (processes) according to the LCFK computed for the specific cluster (in the example, LCFK1 and LCFK2, respectively).



```
• App1 flow key: 5-tuple              [ IP_SRC | IP_DST | PORT_SRC | PORT_DST | PROTO ]

• App2 flow key: 3-tuple              [ IP_SRC | IP_DST |    -     |    -     | PROTO ]

• App3 flow key: TCP/UDP ports        [   -    |   -    | PORT_SRC | PORT_DST |   -   ]

  => Largest Common Flow Key 1        [ IP_SRC | IP_DST |    -     |    -     | PROTO ]
  => Largest Common Flow Key 2        [   -    |   -    | PORT_SRC | PORT_DST |   -   ]
```

Figure 4. Application clustering.

7. An Enif-Lang "Hello World": Tracking and Counting IP Flows

This section presents an example of a simple monitoring application that tracks and counts the different IP packet flows from a traffic source. The objectives of the presentation are manifold: first, to give a practical example of the use of Enif-Lang; second, to show the integration of Enif-Lang within Haskell and its libraries; third, to show the usage of the Enif-Lang own library; and finally fourth, to provide an experimental assessment of the impact of the language processing on the application performance.

```
import Language.Enif.Flow

- defined in Language.Enif.Flow ----------
mkFlowKey5 :: Packet -> FlowKey5
mkFlowKey5 pkt = FlowKey5 byteSwap32 (ipSaddr pkt)
      byteSwap32 (ipDaddr pkt)
      byteSwap16 (srcPort pkt)
      byteSwap16 (dstPort pkt)
      protocol pkt
---------------------------

flowMap :: HashTable FlowKey5 Int
flowMap = newHashTable 1000000

enif_main :: Packet -> ActionIO ()
enif_main pkt = do
  let key = mkFlowKey5 pkt
       e <- lookup flowMap key
      case e of
             Nothing -> insert flowMap key 1
             Just x  -> insert flowMap key (x+1)
```

The program begins with the definition of the associative table `flowMap` that uses the canonical IP 5-tuple of type `FlowKey5` as the indexing key, and contains counters (of `Int` type).

Currently, the `flowMap` table is implemented on top of the `HashTable` Haskell library and is efficiently implemented as a *Cuckoo* hash table [36]. In addition, while the `Int` type is obviously also inherited from the Haskell Language, the `FlowKey5` is instead defined in the `Enif.Flow` library which ships with Enif-Lang.

The hash table is then initialized to 1,000,000 entries. Next, the main body of the program is represented by the function `enif_run` which takes packets (of type `Packet`, yet defined in the `Enif.Flow` library), and produces an I/O action. The function first extracts the canonical IP 5-tuple from the packet, then performs flow `lookup` in the hash table and finally updates the `flowMap` accordingly.

The functions `mkFlowKey5` and `lookup` are both defined in the `Enif.Flow` library, in addition to the types of the packet header. For the sake of clarity, the body of the function `FlowKey5` is also explicitly reported between the comment lines in the code.

Even though very basic, the example shows the expressiveness of the language as well as the simplicity of its usage. First, the extremely strong typedness of the language, which includes highly tailored types of data specifically targeted at packet parsing processing, is worth noticing. As a comparison, to produce the same output by using C/C++ languages would have required a much longer piece of code, including a full handmade packet parsing section. Enif-Lang largely simplifies the whole process of code writing by exposing a rich API that includes a large number of helper functions that, as shown above, dramatically reduce the coding effort and allow even non-expert network programmers to readily access traffic information in a few lines of code.

7.1. Performance Evaluation

The *flow tracker* application described above is next used as a simple benchmark to evaluate the performance that can be reasonably attained by processing traffic with Enif-Lang. The experimental setup consists of a pairs of identical PCs with a 8-core 3.0 GHz Intel Xeon E5-1660V3 on board, Intel 82599 10G NICs and running a Linux Debian distribution with kernel version 4.9. The two servers are connected through a 10 Gbps copper link; one of the two PCs is used as a traffic generator while the second acts as the monitoring probe and runs the flow tracker application.

The traffic generator uses `pfq-gen`, an open-source tool included in the PFQ distribution, to send synthetic UDP packets with randomized IP addresses. The random *depth* is configurable so as to control the total number of different IP flows. In order to evaluate the actual traffic rate that can be sustained by the flow-tracker app—hence, estimate the actual burden of the Enif-Lang program—we used PFQ as the underlying I/O framework. This guarantees that no packets are lost at the interface and that packet drop can only occurs at the application level. More specifically, PFQ was set to use two kernel threads retrieving packets from the network card and delivering to a single instance of the application.

The measurement system was instrumented to give the average number of processed packets over non-overlapped time windows of 1 s. Every run lasted 50 s, hence each experiment produced a time series of 50 samples of measured packet rates.

The results of the first experiment are reported in Figure 5, which depicts the average throughput of the application for increasing values of the flow cardinality in the worst case of minimum packet size of 64 Bytes. The plotted data are computed by averaging the time series of the measured throughput over the 50 s.

Albeit the maximum throughput is attained for the lowest number of flows, the application proves to consistently sustain more than 12 Mpps packet-rate until the table load factor hits 20% (around 200 K flows). For larger number of flows, the performance slowly decreases because of hash collisions that imply relocations within the Cuckoo table. Note that the table can accommodate more than the 1 M entries defined at the initialization time as its implementation is self-resizing.

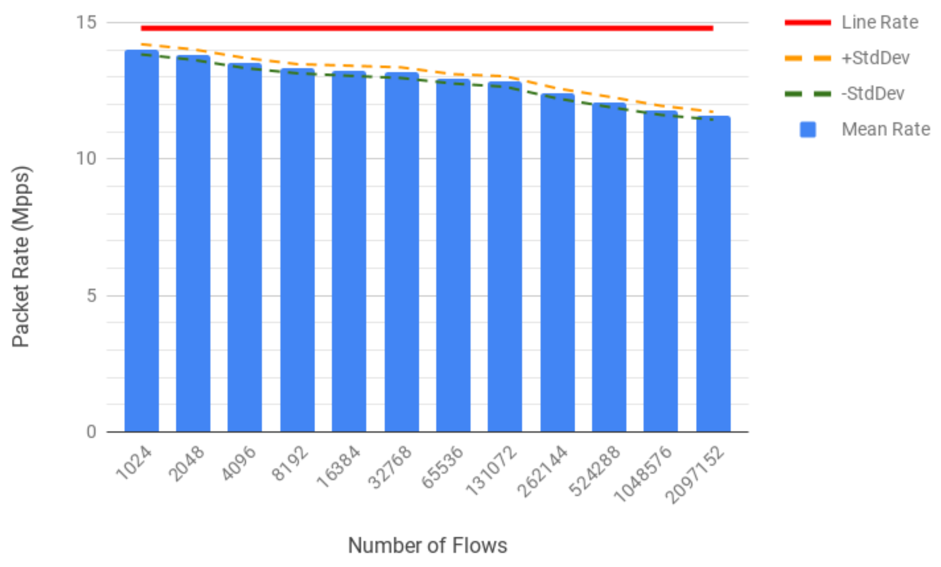


Figure 5. Flow tracker application performance.

The overall impact of the application is therefore reasonably low, with only a 1–2 Mpps rate drop with respect to the theoretical line rate. This level of performance is quite remarkable as the experiment is carried out in the worst case scenario of minimum sized packets, whereas the packet rate is realistically much lower with the typical packet lengths of real traffic. Indeed, when the packet size increases, Figure 6 proves that the application throughput consistently catches up with the input traffic rate by reaching line-rate processing speed even with the largest flow cardinality of more than 2 M flows.

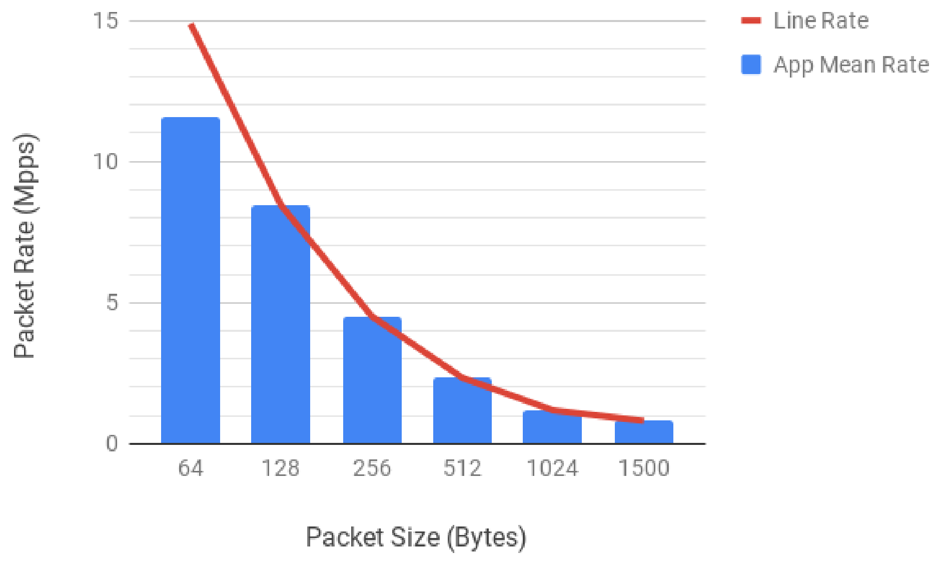


Figure 6. Flow tracker application performance.

In order to investigate the throughput variability, Figure 5 also reports the upper and lower limits given by the sample standard deviations computed over the data. The two curves prove that the attained throughput is vary stable, with a measured coefficient of variation that slightly exceeds 1% in all cases. For the sake of clarity, the whole time-series of throughput samples measured in the case of 1024 input flows is shown in Figure 7, along with the sample mean and standard deviations' range.

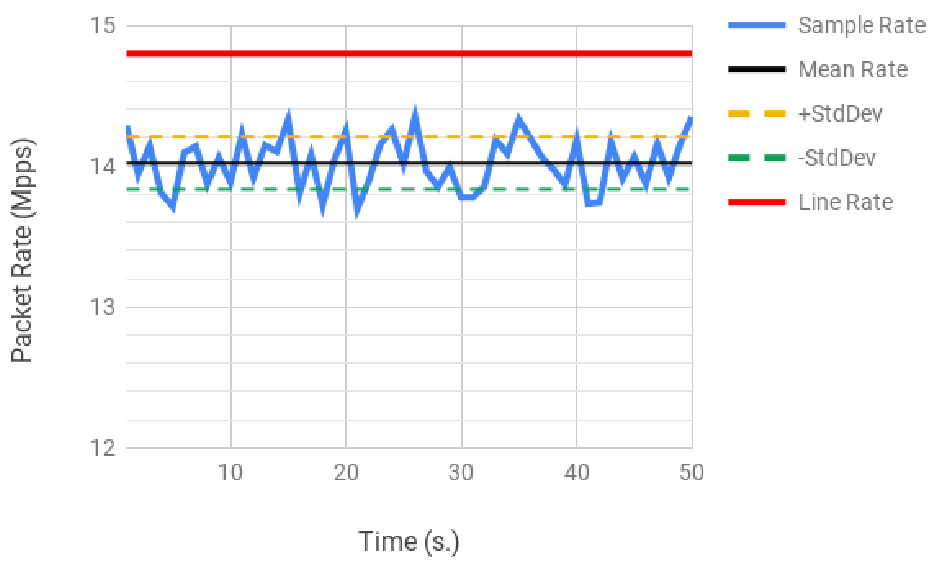


Figure 7. Flow tracker application performance.

8. Use Cases

This section goes further regarding the simple flow tracker monitoring application by presenting the use of Enif-Lang in the development of two slightly more practical applications.

The first application implements a *stateless* load balancer that distributes packets to a cluster of end-nodes, namely network devices or local applications that run a more complex processing, such as Deep Packet Inspection. The second application, instead, is the Enif-Lang implementation of a simple stand-alone simple stateful firewall based on the *port knocking* scheme.

Both cases allow for further pointing out the two-fold use of Enif-Lang to implement (i) a stand-alone network function (load balancing) and (ii) a first stage of traffic elaboration to be followed by a next (possibly multi-threaded) traditional application.

As anticipated in Section 3, in the latter case, packets are distributed to the application through `tun` interfaces, which, in fact, are the endpoints of the Enif-Lang processing. However, while Enif-Lang applications expose (in a declarative way) to the runtime their desired flow key, standard applications do not. Hence, the runtime must be manually instrumented to select the flow keys in order to compute the largest common flow mask and handle the `tun` interfaces accordingly.

8.1. Stateless Processing

DPI applications are typically quite computation expensive and largely benefit from splitting their elaboration across multiple workers (thread/processes). As an example, in [18], the huge performance improvements brought by parallelizing Bro [37] into multiple instances over a multi-core platform have been shown.

However, splitting DPI elaborations over multiple workers requires maintaining flow consistency. In addition, many DPI applications take advantage of DNS packets to build classification trees to improve application recognition. To this aim, the following load-balancer preserves layer 3 symmetric flow consistency by spreading IP packets according to the `steer_p2p` steering function while *broadcasting* DNS packets to all DPI workers. The latter operation sends a copy of all DNS packets to all application workers regardless of the flow definition, so as to allow any computation thread/process to build its own complete DNS tree without the need of accessing data shared among cores, hence ensuring core-isolation.

```
is_dns = has_port 53

enif_main :: Packet -> ActionIO ()
enif_main pkt =
if is_dns pkt
      then broadcast pkt
         else steer_p2p pkt
```

8.2. Stateful Processing

The second use case presents the implementation of the simple port knocking firewall. The problem is well described in [8] and provides a didactically elegant example of a stateful application that blocks all packets with the exception of the ones that *know the locker code* represented by a sequence of four predefined TCP port numbers (5123, 6234, 7345 and 8456 in the example).

The example uses two tables. The first table contains the list of the currently authorized flows and is implemented as an associative map based on the classic 5-tuple key. The second table, instead, is indexed by the 3-tuple keys, and implements the state machine for tracking the knocking sequence. Any time the expected destination port is found, the state is updated through the function `next_if` until it reaches the state value 4 which opens the firewall and the corresponding flow is authorized upon its insertion into the `auth_flow` table.

```
auth_flow :: HashTable FlowKey5 Int
auth_flow = newHashTable 1000000

knock_table :: HashTable FlowKey3 Int
knock_table = newHashTable 1000000

next_if pred key3 value =
    if pred
     then insert knock_table key3 value
     else insert knock_table key3 value

enif_main :: Packet -> ActionIO ()
enif_main pkt = do
let k5 = mkFlowKey5 pkt
    f5 <- lookup auth_flow k5
    case f5 of
     Just _  -> kernel pkt
        Nothing -> do
            let k3 = mkFlowKey3 pkt
            f3 <- loolup knock_table k3
            case f3 of
             Nothing -> next_if (dst_port pkt == 5123) k3 1 >=> drop
                Just 1  -> next_if (dst_port pkt == 6234) k3 2 >=> drop
                Just 2  -> next_if (dst_port pkt == 7345) k3 3 >=> drop
                Just 3  -> next_if (dst_port pkt == 8456) k3 4 >=> drop
                Just 4  -> do
                 when (dst_port pkt == 22) $ do
                     delete knowk_table k3
                        insert auth_flow k5 1
                        kernel pkt
```

9. Conclusions

The paper presents Enif-Lang, a functional language for developing network functions on multi-core middleboxes based on commodity hardware. The language is implemented as an embedded DSL of the Haskell functional language and is specifically targeted at easing network softwarization by providing a compact but expressive formal description of a generic processing machine for high performance network traffic manipulation. The programming model explicitly addresses automatic parallelism through the functional paradigm of packets immutability as well as through the concept of application clustering for traffic splitting across multiple computation resources. A few examples are reported in the paper to exemplify the use of the language in practice, as well as its baseline performance. Two simple use cases are also given to show the expressiveness of the language in handling both stateless and stateful packet processing.

Author Contributions: The work presented here was carried out in collaboration among all authors. Conceptualization, N.B., S.G. and G.P.; Methodology, N.B. and G.P.; Software, N.B.; Validation, N.B. and G.P.; Writing—Original Draft Preparation, N.B. and G.P.; Writing—Review & Editing, G.P.; Funding Acquisition, S.G. and G.P.

Funding: This research was partly funded by the Regione Toscana project OASI, grant number 7429.31052017.113000046, and by the Università di Pisa PRA project CONCEPT, grant number PRA_2018_11.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fusco, F.; Deri, L. High speed network traffic analysis with commodity multi-core systems. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC'10) , Melbourne, Australia, 1–30 November 2010; pp. 218–224.
2. Deri, L. PF_RING ZC (Zero Copy). Available online: http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/ (accessed on 15 June 2018).
3. Rizzo, L. Netmap: A novel framework for fast packet I/O. In Proceedings of the 2012 USENIX Annual Technical Conference, Boston, MA, USA, 13–15 June 2012; USENIX Association: Berkeley, CA, USA, 2012; pp. 1–12.
4. DPDK. Available online: http://dpdk.org (accessed on 15 June 2018).
5. Bonelli, N.; Giordano, S.; Procissi, G. Network Traffic Processing With PFQ. *IEEE J. Sel. Areas Commun.* **2016**, *34*, 1819–1833. [CrossRef]
6. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
7. Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
8. Bianchi, G.; Bonola, M.; Capone, A.; Cascone, C. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 44-51. [CrossRef]
9. The BEBA (Behavioral Based Forwarding) H2020 EU Project. Available online: https://cordis.europa.eu/project/rcn/194157_en.html (accessed on 15 June 2018).
10. Bonelli, N.; Giordano, S.; Procissi, G.; Abeni, L. A Purely Functional Approach to Packet Processing. In Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'14), Los Angeles, CA, USA, 20–21 October 2014; ACM: New York, NY, USA, 2014; pp. 219–230.
11. Bonelli, N.; Giordano, S.; Procissi, G. A pipeline functional language for stateful packet processing. In Proceedings of the 2017 IEEE Conference on Network Softwarization (NetSoft), Bologna, Italy, 3–7 July 2017; pp. 1–4.

12. Braun, L.; Didebulidze, A.; Kammenhuber, N.; Carle, G. Comparing and improving current packet capturing solutions based on commodity hardware. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC'10), Melbourne, Australia, 1–30 November 2010; ACM: New York, NY, USA, 2010; pp. 206–217.
13. Moreno, V.; Ramos, J.; Santiago del Rio, P.; Garcia-Dorado, J.; Gomez-Arribas, F.; Aracil, J. Commodity Packet Capture Engines: Tutorial, Cookbook and Applicability. *IEEE Commun. Surv. Tutor.* **2015**, *17*, 1364–1390. [CrossRef]
14. Gallenmüller, S.; Emmerich, P.; Wohlfart, F.; Raumer, D.; Carle, G. Comparison of Frameworks for High-Performance Packet IO. In Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'15), Oakland, CA, USA, 7–8 May 2015; IEEE Computer Society: Washington, DC, USA, 2015; pp. 29–38.
15. SolarFlare. OpenOnLoad. Available online: http://www.openonload.org (accessed on 15 June 2018).
16. Moreno, V.; Río, P.M.S.D.; Ramos, J.; Dorado, J.L.G.; Gonzalez, I.; Arribas, F.J.G.; Aracil, J. Packet Storage at Multi-gigabit Rates Using Off-the-Shelf Systems. In Proceedings of the 2014 IEEE International Conference on High Performance Computing and Communications (HPCC'14), Paris, France, 20–22 August 2014; IEEE Computer Society: Washington, DC, USA, 2014; pp. 486–489.
17. Marian, T.; Lee, K.S.; Weatherspoon, H. NetSlices: Scalable multi-core packet processing in user-space. In Proceedings of the 2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Austin, TX, USA, 29–30 October 2012; pp. 27–38.
18. Bonelli, N.; Vigna, F.D.; Giordano, S.; Procissi, G. Packet Fan-Out Extension for the pcap Library. *IEEE Trans. Netw. Serv. Manag.* **2018**. [CrossRef]
19. Linux Kernel Contributors. PACKET_MMAP. Available online: https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt (accessed on 15 June 2018).
20. Han, S.; Jang, K.; Park, K.; Moon, S. PacketShader: A GPU-accelerated software router. In Proceedings of the ACM SIGCOMM 2010 Conference on SIGCOMM (SIGCOMM'10), New Delhi, India, 30 August–3 September 2010; ACM: New York, NY, USA, 2010; pp. 195–206.
21. Egi, N.; Greenhalgh, A.; Handley, M.; Hoerdt, M.; Huici, F.; Mathy, L.; Papadimitriou, P. Forwarding path architectures for multicore software routers. In Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO'10), Philadelphia, PA, USA, 30 November 2010; ACM: New York, NY, USA, 2010; pp. 1–6.
22. Morris, R.; Kohler, E.; Jannotti, J.; Kaashoek, M.F. The Click modular router. *SIGOPS Oper. Syst. Rev.* **1999**, *33*, 217–231. [CrossRef]
23. Dobrescu, M.; Egi, N.; Argyraki, K.; Chun, B.; Fall, K.; Iannaccone, G.; Knies, A.; Manesh, M.; Ratnasamy, S. RouteBricks: Exploiting parallelism to scale software routers. In Proceedings of the Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (ACM SIGOPS), Big Sky, MT, USA, 11–14 October 2009; ACM: New York, NY, USA, 2009; pp. 15–28.
24. Rizzo, L.; Carbone, M.; Catalli, G. Transparent acceleration of software packet forwarding using netmap. In Proceedings of the 2012 IEEE INFOCOM, Orlando, FL, USA, 25–30 March 2012; pp. 2471–2479.
25. Barbette, T.; Soldani, C.; Mathy, L. Fast Userspace Packet Processing. In Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'15), Oakland, CA, USA, 7–8 May 2015; pp. 5–16.
26. Sun, W.; Ricci, R. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In Proceedings of the Architectures for Networking and Communications Systems 2013 (ANCS'13), San Jose, CA, USA, 21–22 October 2013; IEEE Press: Piscataway, NJ, USA, 2013; pp. 25–36.
27. Bonelli, N.; Procissi, G.; Sanvito, D.; Bifulco, R. The acceleration of OfSoftSwitch. In Proceedings of the 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Berlin, Germany, 6–8 November 2017; pp. 1–6.
28. Huici, F.; others. Blockmon: A high-performance composable network traffic measurement system. *SIGCOMM Comput. Commun. Rev.* **2012**, *42*, 79–80. [CrossRef]
29. SnabbCo. Snabb Switch. Available online: https://github.com/SnabbCo/snabbswitch (accessed on 15 June 2018).

30. Bonola, M.; Bifulco, R.; Petrucci, L.; Pontarelli, S.; Tulumello, A.; Bianchi, G. Implementing advanced network functions for datacenters with stateful programmable data planes. In Proceedings of the 2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), Osaka, Japan, 12–14 June 2017; pp. 1–6.
31. Monsanto, C.; Reich, J.; Foster, N.; Rexford, J.; Walker, D. Composing Software-defined Networks. In Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13), Lombard, IL, USA, 2–5 April 2013; USENIX Association: Berkeley, CA, USA, 2013; pp. 1–14.
32. The Frenetic Project. Available online: http://frenetic-lang.org/ (accessed on 15 June 2018).
33. de Bruijn, W.; Bos, H.; Bal, H. Application-Tailored I/O with Streamline. *ACM Trans. Comput. Syst.* **2011**, *29*, 1–33. [CrossRef]
34. VPP. Available online: https://wiki.fd.io/view/VPP (accessed on 15 June 2018).
35. Linux Enhanced BPF (eBPF) Tracing Tools. Available online: http://www.brendangregg.com/ebpf.html (accessed on 15 June 2018).
36. Pagh, R.; Rodler, F.F. Cuckoo hashing. *J. Algorithms* **2004**, *51*, 122–144. [CrossRef]
37. The Bro Network Security Monitor. Available online: https://www.bro.org (accessed on 15 June 2018).