Efficient Handling of Protocol Stacks for Dynamic Software Packet Processing

Dominik Scholz, Paul Emmerich, and Georg Carle
Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich
{scholzd|emmericp|carle}@net.in.tum.de

***Abstract*—New standards for 40 and 100 Gbit/s and beyond impose increasing demands for software packet processing frameworks. Protocol stacks used in high-speed packet processing frameworks are reduced to basic functionality to cope with the performance requirements. Development of sophisticated applications require manifold functionality from the protocol stack, while being flexible and extensible to allow support of newly developed protocols. This becomes increasingly important as for instance data-center operators apply more forms of packet encapsulation, generating complex protocol stacks. Existing examples for high-performance frameworks are not able to fulfill all these aspects or have to make compromises.**

We present the protocol stack of the libmoon framework, a novel design for dynamic protocol stacks based on code generation and JIT compilation. Defining the format of a new protocol header is the only action that has to be performed by a programmer. The desired protocol stack is generated automatically from this information, allowing combination of arbitrary headers. Thereby, a complex and full set of utility functions is provided to the application developer. We evaluate our protocol stacks using measurements in a VXLAN setup, which also highlight how the new protocol stack API allows to easily create applications in a flexible and intuitive manner.

I. Introduction

The continuous demand for increasing performance raises the requirements for the soft- and hardware of networking devices. With the standardization of 40 and 100 Gigabit Ethernet in recent years [1], devices have to be capable of managing multiples of this load. Several software packet processing frameworks have been developed to meet these demands [2], [3]. They utilize new and specialized techniques to circumvent traditional bottlenecks of networking applications. A common approach is to bypass the conventional network stack of the operating system (OS) and do all packet related operations themselves [4], [3]. While this results in a loss of features provided by the OS, performance can be gained as all unnecessary and inefficient processing steps can be removed or replaced with optimized solutions.

However, in all these frameworks the core functionality is processing packets at a low level. The protocol stack that employs the protocols of the different levels of the ISO/OSI model and allows to easily modify packets is important for the developers experience and consequently the overall deployment of the framework and application. A wide spread of different protocols with unique usage scenarios and varying complexity and requirements makes the implementation of a proper protocol stack that fulfils the requirements for a packet processing framework difficult, especially as the overall performance is the key demand and must not be impeded. Therefore, the protocol stack is usually reduced to basic functionality or compromises are made in regards to utility, flexibility and extensibility. This paper focuses on providing a user-friendly programming interface to efficiently access header fields in complex protocols. Implementations of high-level protocol semantics beyond header validations are beyond the scope of this paper.

libmoon is a packet processing framework based on DPDK [5]. In previous work it was shown that libmoon can be used to create applications for manifold networking tasks, including the packet generator MoonGen. The downside of working with packets in libmoon and MoonGen was that packets had to be crafted by setting every byte of them manually. While clearly this is fast, it is prone to errors, and neither flexible nor reusable as, for instance, the programmer has to take care of correct byte order and calculating all offsets within the packet. For every new application, the complete protocol stack has to be crafted again, resulting in a large amount of duplicated code as protocols are reused and several protocol fields usually are set to the same standardized value.

We present the high-performance flexible protocol stack that we designed and implemented for libmoon. It is based on dynamic code generation and JIT compilation. We show that our dynamic concept of generating the protocol stack offers maximum utility while allowing for easy extensibility to accommodate future protocols. The effort for the programmer is thereby reduced to a minimum. To verify that the implemented solution fulfills the performance requirements, we evaluate the performance of our generated code in a VXLAN encapsulation task and compare it with a hand-coded version.

This paper is structured as follows. In Section II a survey of existing protocol stacks in modern software packet processing frameworks is presented. The architecture of libmoon and it's packet IO backend DPDK and their impact on the protocol stack are discussed in Section II-B. Section III illustrates the requirements, chosen architecture and actual implementation of the developed protocol stack. The implemented protocol stack is evaluated in terms of performance and flexibility while also highlighting its usability in Section IV, using VXLAN as example task. We conclude with a summary in Section V.

II. Background

Frameworks, tools, and applications for software packet processing at rates up to and beyond 10 GbE such as DPDK, PF_RING, or PFQ allow the user to modify packets in one way or another.

A. Related work

The term protocol stack often refers to fully featured implementations of protocol semantics. One example is the networking stack found in operating systems, featuring a large variety of supported stacks. The aforementioned frameworks come without support for such a stack. Specialized userspace stacks like mTCP [6] for DPDK exist that fill this gap. However, such implementations are not our focus here, we only discuss representing and addressing packet headers on a lower level as provided by the framework.

DPDK comes with struct definitions and helper functions for common protocols that can be used to build stacks manually. Pktgen-DPDK [7], a packet generator, is an interesting case study for the representation of protocol stacks. It supports generating traffic with common protocols such as UDP, TCP, ARP ,and ICMP. Moreover, it can generate traffic encapsulated with the GRE (Generic Routing Encapsulation) tunneling protocol, a feature requiring a more sophisticated approach if implemented properly.

The utility functions per protocol are reduced to filling the header based on a sequence object as shown in Listing 1. This centralized object for the whole protocol stack contains a list of keywords reflecting certain fields like the IP source address field. Other header fields like the IP version are preset and hard-coded to fixed values and cannot be customized. Furthermore, Pktgen offers a scripting interface, which allows to define streams of packets during runtime.

```
local seq_table = {
        ["eth_dst_addr"] = "0011:4455:6677",
        ["eth_src_addr"] = "0011:1234:5678",
        ["ip_dst_addr"] = "10.12.0.1",
        ["ip_src_addr"] = "10.12.0.1/16",
        ["sport"] = 9,
        ["dport"] = 10,
        ["ethType"] = "ipv4",
        ["ipProto"] = "udp",
        ["vlanid"] = 1,
        ["pktSize"] = 128
};
pktgen.seqTable(0, "all", seq_table );
```

Listing 1: Pktgen's sequence table for central packet configuration[1]

Generating a full protocol stack of for instance an ICMP Echo request is only implemented by hard-coding the sequence of the respective headers. This is even more cumbersome for GRE as each type, based on either IP or Ethernet, is hard-coded separately. Clearly, this approach does not scale well when more protocols are added.

The PFQ [8] networking framework focuses on creating applications that make use of "in-kernel functional processing and packets steering across sockets/end-points" [8] using C, C++, Haskell or their own *pfq-lang* domain-specific language (DSL). The C++ interface offers no support for packet headers, e.g., the packet generator tool[2] creates ICMP packets by manually setting all bytes. The functional language intended for packet processing allows to query, filter and modify the properties of the packet, but is limited to simple stacks with monotonically increasing layers, i.e., tunneling protocols are not supported.

PF_RING ZC [9] is a network socket intended for capturing, filtering and analyzing packets in high-performance environments. The sample packet generator application[3] defines its own header structures. These are then filled member by member, all values are hand-crafted and no utility functions are used, especially for instance for the calculation of the IP checksum. Furthermore, the offsets and header sizes have to be calculated manually when allocating the structures. However, PF_RING also supports to send packets from pcap files. The suggested work-flow is to generate the traffic as a pcap with another tool that allows for easy modification and crafting of packets and then sending it with PF_RING.

The netmap [10] kernel module designed for fast, but also safe packet I/O uses standard system calls, making it easy to modify existing raw socket or libpcap applications to work on top of netmap [4]. The sample packet generator application[4] shows that common C structures are used to build packets. No further utility functions are provided.

The multi-platform traffic generator and analyzer Ostinato [11] is different compared to the previously introduced frameworks as it focuses on providing a powerful graphical interface for the protocol stack to craft, modify and analyze packets. While Ostinato is able to directly send crafted packets, the primary intent is to create or edit pcap files which can then be replayed by a processing framework intended for high-performance. The tool supports all common protocols, a variety of tunneling protocols and also higher level, text-based protocols like HTTP. Not only can these protocols be stacked in any order, but also every header field can be set via the GUI to user defined values. All fields are initially set to intelligent defaults, values that for instance depend on the length of the packet like the IP length field are automatically calculated, but can be overwritten manually, too. The option to generate streams of packets with a continuously changing member value is also provided. This is a very useful implementation for a packet generator frontend, but unsuitable for general-purpose packet processing.

Snabb [12] is another project building on a userspace driver to speed up packet processing. As with libmoon, it uses Lua as programming language in combination with LuaJIT to offer the user a simple, yet fast, scripting environment, easing the development of applications. Snabb provides a protocol stack *lib.protocol* that offers versatile utility functions to manipulate headers and craft packets. In general, an unknown packet can

[1] Excerpt taken from http://pktgen.readthedocs.org/en/latest/index.html

[2] https://github.com/pfq/PFQ/blob/master/user/tool/pfq-gen.cpp

[3] https://github.com/xtao/PF_RING/blob/master/userland/examples/pfsend.c

[4] https://github.com/luigirizzo/netmap/blob/master/apps/pkt-gen/pkt-gen.c

be parsed, which as a result returns a list of the headers representing the full protocol stack. Each header can then be manipulated using setter and getter functions for the fields. On the other hand, a full protocol stack even for a complex (tunneling) packet can be crafted. Listing 2 shows how a tunneling protocol stack is built and configured. However, because of performance issues, these utility functions cannot be used for performance critical parts of the program [13].

```
o._encap = {
  ipv6  = ipv6:new({ next_header = 47, hop_limit = 64,
    src = conf.local_vpn_ip, dst = conf.remote_vpn_ip}),
  gre   = gre:new({ protocol = 0x6558,
    checksum = conf.checksum, key = conf.label })
o._encap.ether = ethernet:new({ src = conf.local_mac,
    dst = conf.remote_mac or
          ethernet:pton('02:00:00:00:00:00'),
    type = 0x86dd })
}
```

Listing 2: Creating a GRE encapsulated packet with Snabb[5]

None of the introduced processing frameworks combines the aspects of fast performance, flexibility, usability and extensibility in one protocol stack. This is either because the tool is not focusing on packet modifications or a compromise between performance and utility is being done favoring the former. However, three interesting features that a protocol stack should offer can be identified: Firstly, utility functions for data types like IP addresses need to be provided. Secondly, the usage of one centralized table-like structure that makes use of labels to reference protocol fields provides high usability. Within one structure, the complete configuration of a packet should be done, which in combination with meaningful names, constants, and default values makes it intuitive and easy to use. Lastly, the protocol stack should not hinder the developer, i.e., total freedom should be granted to the extent that even illegal packets should be able to be crafted.

B. The libmoon framework

libmoon is a framework for building packet processing applications in the scripting language Lua [14], [5]. It combines the userspace packet processing framework DPDK [15] with the Lua just-in-time (JIT) compiler LuaJIT [16]. Using a high-level language allows for short development cycles of packet processing applications. Lua code compiled with LuaJIT is as fast as equivalent code written in C and can embed existing C code. This is achieved by using low-level C structs instead of the default Lua data structures in all performance-critical paths. The disadvantage of this is that the usual safety features of scripting language like memory-safety are not available in these paths. However, the critical code is handled by the framework, not by the user application.

Figure 1 shows a high-level overview of the architecture of libmoon. The whole functionality is controlled by the *userscripts* which consist of two main parts: setup and runtime. It first configures all used network cards in the master task in the *setup phase* and then starts several completely independent

[5]Excerpt taken from https://github.com/SnabbCo/snabbswitch/blob/master/src/apps/vpn/vpws.lua

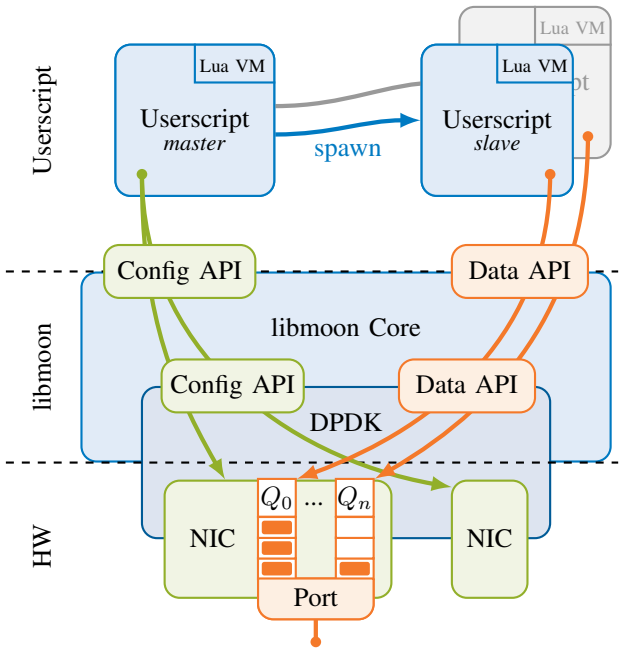


Figure 1: libmoon's architecture (adapted from [5])

slave tasks that handle the packet processing tasks. Each task separately has to allocate its required memory pools for packet buffers. All buffers within one pool are initialized with the same values as set by the user, allowing to define a template of the desired packet, already before the application starts processing packets. During the actual runtime of the application, a prefilled packet buffer can be retrieved from the pool and further modifications per packet can be applied. All tasks have access to extensive utility libraries such as the protocol stack described in this paper.

III. Dynamic Protocol Stack

The protocol stacks of existing frameworks surveyed in Section II are static, meaning that the specific required stack has to be build by the programmer, in most cases from scratch. The advantage is that they can easily be optimized by an ahead-of-time compiler. However, the drawbacks are manifold, as becomes apparent when looking at modern tunneling protocols like VXLAN. Two different protocol stacks, one for each en- and decapsulated packets, are required. With an inflexible approach, both have to be hardcoded separately yielding code duplication, although a large amount of structure and bytes, the decapsulated packet, is shared by both stacks. Further, the data of that part of packets does not have to be modified at all, as merely headers are prepended. A related problem is that higher layer protocols in the ISO/OSI model can be based on different protocols underneath, a prominent example being IPv4 and IPv6 on the network layer. In most of the frameworks presented in Section II, if the user wants to create a UDP packet once based on IPv4 and once on IPv6, he would be required to implement both protocol stacks separately, for all layers and from scratch. This duplication of code continues

even further when adding new protocols, generating more and complex stacks that have many layers in common.

The requirements for the dynamic protocol stack of the libmoon framework are as follows.

Req. 1 – Performance The protocol stack must not reduce the processing performance of the framework, i.e., packets have to be processed potentially at line-rate of 10 GbE or beyond. In other words, an equal performance level as if the task would have been implemented with low level operations has to be achieved.

Req. 2 – Flexibility While granting the user full customizability to create even malformed packets, the protocol stack must be flexible to allow complex operations like packet en- or decapsulation with few operations and low computational overhead.

Req. 3 – Usability The API of the protocol stack must be easy to understand and intuitive to work with. It must offer functions providing utility to the user by obsoleting repetitive tasks. This includes data type and byte-order conversions, as well as generating complete, prefilled and legal packets per default.

Req. 4 – Extensibility and composability The protocol stack must be modular, i.e., every protocol header is implemented by itself, allowing to easily add new protocols in the future. Individual headers must be combinable to complex stacks. At the same time, this process should be easy and automated, requiring only minimal developing effort whenever possible.

The survey of existing packet generators has shown that especially <**Req. 1**> and <**Req. 3**> pose a challenge, leading to a trade-off between performance and usability. The following sections outline the concept and architecture of libmoon's protocol stack which not only unifies these contradictory points, but moreover fulfills all listed requirements.

A. Conceptual overview

Not all functions need to be fast: functions used during the setup phase are allowed to be slow. Only the startup time of the application will be delayed, which is not a time-critical aspect. In turn, these functions offer as much utility as possible during this phase, generating templated packets with even one line of code. During the actual runtime of the application, all called functions are critical to the performance. All functions used there primarily need to be fast.

The only actual implementation effort is defining the layout of a concrete protocol, i.e., defining the syntactical structure of the header. However, this has to be done only once, i.e., for protocols or headers that libmoon does not yet support. The complete generation of a protocol stack is performed automatically and dynamically through JIT compilation on demand. Users can define their desired protocol stack by defining which header should appear in what order. This concept follows and is based on the ideas of the ISO/OSI model. Each layer, in this case protocol, is independent and self-contained from the next layer, wherefore each protocol header can be implemented separately.

B. Statically implementing a new header

libmoon's protocol stack requires only the minimum of information about a protocol header, which has to be implemented once. Primarily, this includes the structure of the header, i.e., its members and their bit-size. In addition, semantic information, including how, based on the data of the header, for instance its own size or the next following header can be determined. This process is automated as far as possible and only requires further code changes through interaction with the developer when using data types with additional semantics like IP addresses. Three steps are necessary: defining the header layout, generating wrapper functions for all members and creating functions called on the header as a whole.

Header structure The format of a header is defined as if it was a C *struct* object, consisting of the data types and names for each member. In fact, libmoon creates an actual C structure based on this information of the header using LuaJIT's Foreign Function Interface (FFI). Data types can be basic ones, or specifically created as is the case for instance for MAC or IP addresses. Members with variable size have to be implemented as variable-length array members, the header is duplicated and specialized for each distinct header length. Furthermore, the name of such a member has to be marked as variably sized in the headers Lua object. This allows for special treatment within the complete protocol stack and to create concrete, fixed sized instances of the header, depending on the current data of a packet. All members have to be aligned correctly within the header using the *packed* attribute; this C structure is later used directly for packet data.

Wrapper for all members To prevent working with low-level data types, wrapper *setter and getter* functions for all members are created in Lua. These functions provide a first level of utility as data type conversions, correct byte order and other data type problems are taken care of. Internally, libmoon creates a Lua metatable object for the underlying C structure of the header and adds all functions to it based on that information. This process is automated for standard data types like integers. Only if the automatically generated utility functions are not sufficient for instance because it is a custom data type, the set and get function has to be manually defined by overwriting the generated one.

As these functions are merely thin wrappers in Lua they fulfill <**Req. 1**>. However, they already provide utility as they take care of low-level interactions and consequently also fulfill <**Req. 3**>. As a result, these functions are fast while providing a minimum of usability and are therefore the functions that should be used during the actual runtime of libmoon to modify packets on a per packet basis.

Functions for the complete header When creating the meta object for a new header, libmoon also adds several templated functions, increasing the abstraction level from mere bytes to not only basic setter and getter functions, but also to functions that perform tasks on the complete header. Their purpose is two-fold: Firstly, functions that provide utility for the user in form of a set and get function for the complete

header. In the case of the set function, the passed argument is a table of labels, each referencing a concrete member of this header. Within the function, for each member the respective set function is called and provided the passed argument or, in case it was not specified, a default value defined once by the developer. This way, using one function call, the complete header is filled with standard default values, while each single member can be customized.

Because of the usage of Lua tables, this function comes at the cost of a significant performance loss. At this abstraction level, performance is not the primary concern, instead, usability on a broader range to fulfill <**Req. 3**> becomes important. Hence, this function belongs to the group of methods that should only be used before the runtime of libmoon to pre-allocate packet buffers when initializing a templated memory pool.

The second set of functions is used to provide semantical information about the header in the context of a complete protocol stack. This includes how to resolve the next following header, how default parameters change (e.g., IPv4 protocol), how the size of the variable member is determined (e.g., the size of the IPv4 options field depends on the Internal Header Length member), or how the sub-protocol type can be determined (e.g., Ethernet headers might have a VLAN tag, TCP headers might have options, etc.). If one or multiple of these case applies to the concrete protocol, the developer has to overwrite the automatically generated empty function to define the semantic information.

The end result of this step is that a new protocol, its layout and required semantical information, is defined so that a concrete instance of it can be used within a complete protocol stack.

C. JIT-compiled protocol stack

The implemented protocols can be used by an application developer to generate arbitrary protocol stacks dynamically on demand. This is possible as the implemented header information is sufficient to fully specify one layer of a protocol stack. From the point of an application developer using the library of available protocols, merely the order of protocols in the concrete required stack has to be defined. Internally, libmoon performs multiple steps to dynamically offer complex utility functions for the whole stack, which will be explained in the following.

Defining a protocol stack Creating a new protocol stack can be performed with one function call of `createStack(args)`, passing as argument a list of protocols using the simple and intuitive embedded DSL defined in Listing 3.

```
<stack>    ::= <protocol> | <protocol>, <stack>
<protocol> ::= <header> | "{", <header>, [",name=" <str>],
               [", subType=" <str>], [", length=" <int>], "}"
<header>   ::= "eth" | "ip4" | "ip6" | "udp" | ...
```

Listing 3: DSL to define a protocol stack

In the simplest case, it defines the order of required protocols, starting with the lowest layer. Syntactic sugar is added by allowing to specify a table instead, which allows for optional arguments to cope with special scenarios. In case a protocol is used multiple times within a stack it has to be uniquely labeled, which can be done with the optional `name` attribute. For protocols that may have different subtypes or protocol layouts, e.g., Ethernet with and without a VLAN tag, the type can specified with the `subtype` argument. Lastly, for headers which can be variably sized, the length of the variably sized member as defined when implementing the protocol can be specified with `length`.

A concrete example for a realistic and complex protocol stack as was observed in a data center environment is illustrated in Listing 4.

```
-- create protocol stack
local asFooStack = createStack(
        {"eth", subType="vlan"}, "ip4", "udp", "vxlan",
        {"eth", name="innerEth"}, {"ip4", name="innerIp4"},
        {"udp", name="innerUdp"}, {"sflow", subType = "ip4"}
)

-- cast buffer to protocol stack
local pkt = asFooStack(mbuf)
```

Listing 4: Creating and using a new protocol stack

The end result for the application developer is that a function to cast a packet buffer to the desired stack is returned.

Internal data structure Based on the defined sequence of protocols using the DSL, internally a C structure for the concrete protocol stack is automatically generated. This is illustrated in Listing 5 for the already introduced example.

```
struct __attribute__((__packed__)) stack_<snip> {
        struct eth_vlan_header    eth;
        struct ip4_header         ip4;
        struct udp_header         udp;
        struct vxlan_header       vxlan;
        struct eth_default_header innerEth;
        struct ip4_header         innerIp4;
        struct udp_header         innerUdp;
        struct sflow_ip4_header   sflow;
        union payload_t           payload;
};
```

Listing 5: Dynamically generated and JIT-compiled C structure

The struct's members are the C structures of the respective defined protocols. Unless specifically labeled by the `name` argument, the member is named like the protocol. Depending on the subtype or length, respective structures are dynamically defined and loaded once by the JIT compiler.

Casting a packet buffer to this structure allows to interpret the array of bytes as defined by the structure of the stack and its protocol members. The C object is extended with a Lua metatable to define utility functions for the complete protocol stack. As a Lua metatable object exists for each member of the stack, the respective utility functions per protocol are available at this abstraction level too. The payload struct is added as final member of each protocol stack. Payload is a union consisting of C99 flexible array members of differently sized integers, this allows to access arbitrary bytes beyond the last specified header.

Generated utility functions Functions operating on the complete protocol stack can be grouped into three categories. First, helper functions only used for internal processes. The

second set of functions are again setter and getter, operating on the complete stack. In this case, the set function calls for each included header its respective set function and passes the complete table of labels. To uniquely reference members of different headers, which could even appear multiple times within the stack, the `name` of the concrete header is prepended to each label for its members, as is demonstrated in Listing 6.

```
pkt:fill{
        -- member of the outer Ethernet header
        ethSrc      = "01:02:03:04:05:06",
        -- member of the inner Ethernet header
        innerEthSrc = "0a:0b:0c:0d:0e:0f"
}
```

Listing 6: Referencing stack members

As default values of a header member can change depending on the context of the complete stack, as is the case for instance for the `ether_type` field of the Ethernet header, default values for undefined labels are replaced using the logic implemented per header. This can be based, for instance, on the next following header or the accumulated length of the preceding headers. This information is available at the abstraction level of the complete stack. As a result, even when calling the set function without setting any labels manually, a legitimate packet is always created. Similar, the getter functions per header, in combination with the semantic information how the next following header is detected per protocol, are used to parse and recursively resolve even completely unknown packets as far as possible. Furthermore, the packet can be dumped in a *tcpdump*-like format. These functions, however, are slow because of the use of Lua tables or recursive operations, while providing high utility, and should only be used to generate templates or for debugging.

The last group of automatically generated utility functions for the complete stack are performance critical functions to be used during the actual runtime. This includes the calculation of all contained checksums or only the checksum of a specific header in software[6], or setting all attributes that depend on the size of the complete packet. In this case, the code for the function that is optimized for performance, i.e., does not use Lua tables to iterate all members of the stack, is generated as string and loaded dynamically through JIT compilation during runtime.

Handling of variably sized headers Headers with variable size complicate the generation of a complete stack immensely, as the C structures of the respective protocol cannot be increased on demand. Using an undefined size results in wrong alignment of the following header. Therefore, the only solution is to create a completely new stack for each different size. While this increases the amount of generated stacks it has no negative impact on the application as recasting the packet buffer to another stack costs virtually no performance as demonstrated in the next section. Furthermore, the new stack is comprised of the same utility functions, only slightly adjusted to accommodate the new size.

[6]Whenever possible the offloading features of the NIC should be utilized to gain performance

Creating a new protocol stack is a CPU-intensive task because of the described internal processing. However, for most applications, the protocol stacks of interest are known in advance and can therefore be generated during the setup phase, or new protocol stacks appear seldom during runtime. This dynamic protocol stack meets all requirements. Maximum utility is provided when generating packet templates during the setup phase. At all times, a minimum amount of utility is guaranteed through offering thin Lua wrappers per member of the underlying C structure, fulfilling <**Req. 1**> and <**Req. 3**>. Thereby, single protocol members can be accessed directly on a per-packet basis. For example, Listing 7 shows an IPv6 packet generator with varying IP addresses and TCP ports implement on libmoon. Utility functions for 128 bit arithmetic allow varying the address on a per-packet basis.

```
local baseIP = parseIPAddress("2001:db8::1")
local counterIP, counterPort = 0
while lm.running() do
        --minimum size for eth/ip6/tcp is 74 bytes
        bufs:alloc(70)
        for i, buf in ipairs(bufs) do
                local pkt = buf:getTcp6Packet()
                --increment IP
                pkt.ip6.src:set(baseIP)
                pkt.ip6.src:add(counterIP)
                --increment port
                pkt.tcp:setSrc(basePort + counterPort)
                counterIP = incAndWrap(counterIP, numIPs)
                counterPort =
                        incAndWrap(counterPort, numPorts)
        end
        -- [...]
        queue:send(bufs)
end
```

Listing 7: Accessing and modifying a packet member on a per packet basis

Additional utility functions for common data types used in network applications provide for instance optimized arithmetic operations. Because of the modular approach, implementing each header completely separate, and large effort to automate even this process, the protocol stack can easily be extended with further protocols, meeting <**Req. 4**>. As the protocol stack allows for maximum customization, generating even complex stacks with few lines of code through the use of an intuitive DSL, the flexibility requirement <**Req. 2**> is realized. All these aspects will be demonstrated in the evaluation in the next section.

IV. Evaluation

We use measurements to verify that the JIT-compiled protocol stack retains the required performance level of libmoon. As sample application the Virtual Extensible LAN (VXLAN) protocol is used, as it demonstrates common tasks for modern tunneling protocols. One host is used as Virtual Tunnel Endpoint (VTEP), encapsulating incoming packets by prepending an Ethernet, IPv4, UDP, and VXLAN header. This requires flexible handling of multiple protocol stacks, as well as utility functions to fill in the header information. The host running the VTEP runs Linux kernel 3.7. It is equipped with an Intel Xeon E3-1230 v2 at 3.3 GHz and 8 MB L3 cache[7] and a 82599ES

[7]http://ark.intel.com/products/65732

10-Gigabit SFI/SFP+ Network Connection[8] NIC. The script implementing the task of the VTEP used for the measurements is accessible in our git repository[9]. Excerpts of it are illustrated in the next section, demonstrating the usage of the dynamic protocol stack. A second, directly connected host is generating and sinking traffic using the MoonGen [17] packet generator.

A. Encapsulation sample script

As illustrated in Listing 8, the encapsulation of packets using VXLAN is straight forward. Before actually processing packets, the required protocol stacks are defined and a template of the encapsulating headers is created. This is being done with one function call during which all members that have to be modified are set using labels.

During runtime, every single received packet is looked at as raw data. Only its total size is required to copy all data as payload to the created encapsulating protocol stack. The remaining tasks are to update the size of the buffer and setting the length members of the IPv4 and UDP headers. The latter can be done with only one function call. All used functions are dynamically generated and optimized for the protocol stack. As last step, the NIC is instructed to calculate the checksums before transmitting the packet.

```
-- create stack
local asRawPacket = createStack()
local asVxlanPacket =
  createStack("eth", "ip4", "udp", "vxlan")
-- creation of packet template
local mem = memory.createMemPool(function(buf)
        asVxlanPacket(buf):fill{
                -- the encapsulating headers
                -- define the VXLAN tunnel
                ethSrc="aa:bb:cc:dd:ee:ff",
                ethDst="00:11:22:33:44:55",
                ip4Src="192.0.2.1",
                ip4Dst="192.0.2.254",
                vxlanVNI=1234,
        }
end)
local txBufs = mem:bufArray()
-- [...]
while libmoon.running() do
        local rx = rxQ:tryRecv(rxBufs, 0)
        txBufs:allocN(rx)
        for i = 1, rx do
                -- cast to generic packet
                local rxPkt = asRawPacket(rxBufs[i])
                -- get size of the packet
                local size = rxBufs[i]:getSize()
                -- cast tx template to VXLAN packet
                local txPkt = asVxlanPacket(txBufs[i])
                -- copy rx raw payload to tx packet payload
                ffi.copy(txPkt.payload,rxPkt.payload,size)
                -- add length of added headers to size
                local totalSize = 46 + size
                -- adjust buffer size
                txBufs[i]:setSize(totalSize)
                -- set the IP/UDP length members
                txPkt:setLength(totalSize)
        end
        -- offload checksums
        txBufs:offloadChecksums()
        -- send encapsulated packet
        txQ:send(txBufs)
end
```

Listing 8: Excerpts from the encapsulation task

[8]http://ark.intel.com/products/32207

[9]https://github.com/emmericp/MoonGen/commit/b05a5eac8c08cb98b3064fcc1bf3ebe7899b4874

Performing these operations without utility functions is error prone and cumbersome. It requires knowledge about the header offsets within the complete stack to copy the data to the correct position. While the copy operation itself remains the same, setting the bytes of the encapsulating headers, including addresses, ports and length members, byte by byte is time consuming, prone to errors and requires drastically more lines of code.

B. Encapsulation performance comparison

This first measurement analyzes the maximum throughput and packet rate when encapsulating minimum sized 64 byte packets at line-rate. To have a baseline comparison that allows to better judge the results, the performance of the developed libmoon task is compared to the VXLAN implementation of Open vSwitch (OvS) [18]. This comparison is however not representative of the performance of OvS as it performs significantly more operations compared to the encapsulation task in Figure 8, including matching of packets and looking up and applying respective actions.

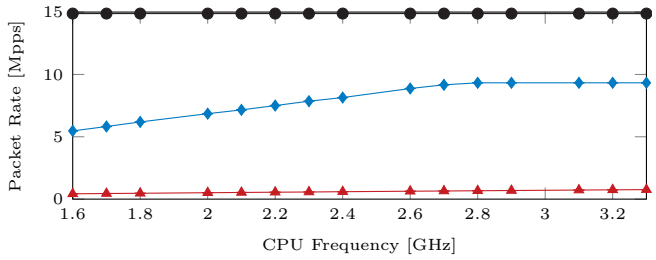
Because of the nature of the VXLAN scenario, the size of received and transmitted packets by the VTEP differ, wherefore both packet rate and throughput are displayed in Figure 2.

libmoon is able to process all incoming packets already at a CPU frequency of 2.8 GHz. The transmitted packets are encapsulated and therefore larger, resulting in not reaching the maximum packet rate of 14.88 Mpps, as the link capacity of 10 Gbit/s is reached first. Before this point the throughput and packet rate increase linearly with the configured frequency. As comparison, OvS, for which the configuration consists of installing one rule specifying the tunnel endpoint, reaches a throughput below 0.5 Mpps. Considering the expensive operations performed, this is consistent with other performance measurements that recorded a maximum rate below 2 Mpps for the simple switching of packets [19]. A final comparison not illustrated in Figure 2 can be made with the packet generator MoonGen, which is based on libmoon and uses the same protocol stack. It is able to saturate a 10 Gbit/s link with minimum sized packets, of which every single one can be customized, already at 1.5 GHz on comparable hardware [5].

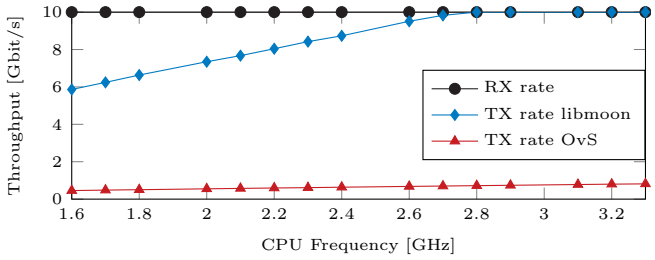
C. Exploiting sequential memory locality

The JIT-compiled utility functions are optimized and produce no new performance overhead, which is demonstrated in the next measurements. In fact, more important is the location of the data within the packet that is being read or modified. Figure 3 illustrates the achieved packet rate at a CPU frequency of 1.6 GHz when encapsulating packets with increasing size, meaning the only difference for the application is the amount of data that has to be copied to the TX buffer.

When copying zero bytes, the performance equals the result of the previous Section, however, it does not decline linearly with increasing packet size and therefore amount of copied bytes. Instead, multiple continuous performance levels can be identified. Within one level, the maximum achieved rate



(a) Packet Rate



(b) Throughput

Figure 2: Peak performance when encapsulating Ethernet frames

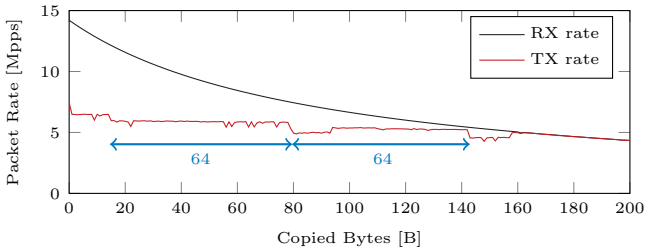


Figure 3: Performance when copying different amounts of consecutive bytes.

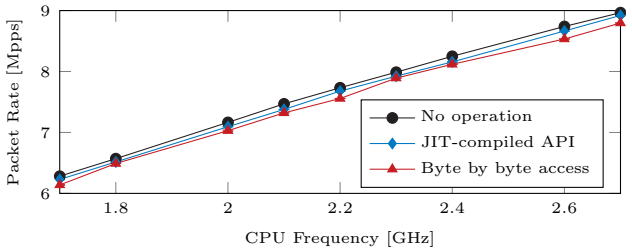


Figure 4: Performance of the old versus the new API when setting the IP address of a packet

decreases only marginally by about 0.1 Mpps. At the end of a level, however, a significant performance loss of roughly 0.5 Mpps can be observed. As indicated in Figure 3 the levels have a size of 64 byte, indicating a correlation with the cache line size. Packet data is an array of sequential bytes, which the dynamically generated protocol stack only casts to a different structure, retaining memory locality. A huge performance loss therefore only appears when accessing data in a different cache line, generating a cache miss. Actual data manipulation within that line only costs cycles depending on the performed operation.

D. Low-level byte access vs. JIT-compiled utility functions

The last measurement compares the performance when performing a typical operation on the packet using the new protocol stack API and its dynamically generated and JIT-compiled utility functions to the old one, which required setting the bytes of the packet manually. The concrete task is to set the source IP address of the packet. For both APIs the packet has to be cast, once as an array of bytes and once as an IP packet type, using the generated IP stack. Then the address is set: all four bytes manually, or, when using the dynamically generated protocol stack via the set utility function of the header's member.

The measurement displayed in Figure 4 is performed for different CPU frequencies when offering the application minimum sized packets at line rate. It not only shows that the total performance loss with less than 0.2 Mpps for this operation is very small and can be explained with the results of the previous section, but also that the new dynamically generated API using wrapper functions yields better results. While a repetition of this experiment yielded the same result, the performed operations on the packets are basic and not complex. Still, it demonstrates that the implemented protocol stack does not affect the performance and provides utility to a developer. The explanation for this result is that the new API can be optimized by the LuaJIT compiler, compared to low-level operations performed byte by byte.

Even for complex applications and operations performed for instance for a TCP SYN proxy based on the libmoon framework[10], the LuaJIT compiler is able to cope and optimize the code of the protocol stack and its utility functions.

V. Conclusion

We presented the dynamic protocol stack of the libmoon framework. It provides high level utility functions to perform packet modifications by accessing header fields, while maintaining the performance of direct low level byte operations. The user-friendliness and flexibility is achieved through automatically created and JIT-compiled code. The API of the protocol stack in combination with the Lua scripting language lowers hurdles for developers creating new software-based networking applications.

The library of existing protocols can be extended with minimal effort: utility functions are generated automatically and newly implemented protocols can be integrated with provided headers. The structure and low-level semantics of each protocol header is implemented completely separately, ensuring modularity. New headers are available immediately

[10]https://github.com/scholzd/MoonCookie

to be used in a custom made protocol stack, which can be defined using a DSL. Because of its flexibility, this allows the creation of even complex stacks with multiple layers, including tunneling and other encapsulating protocols, as presented in this paper.

We have shown that in comparison to the state of the art, a wide-ranging set of utility functions, which are even automatically generated without the need for tedious manual implementation, can be provided, without generating overhead when processing packets. In fact, the JIT compiler is able to optimize the dynamically generated code such that even complex operations are performed as if manual low-level byte by byte manipulations were used instead.

REFERENCES

[1] D. J. Law, A. Healey, P. Anslow, S. B. Carlson, and V. Maguire, "IEEE 802.3bm-2015," 2015.

[2] J. L. Garcia-Dorado, F. Mata, J. Ramos, P. M. S. del Rio, V. Moreno, and J. Aracil, "High-Performance Network Traffic Processing Systems Using Commodity Hardware." Springer Verlag, 2013, pp. 3–27.

[3] S. Gallenmller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet io," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2015)*, Oakland, CA, USA, may 2015, http://www.net.in.tum.de/fileadmin/bibtex/publications/papers/gallenmueller_ancs2015.pdf.

[4] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *USENIX Annual Technical Conference*, 2012, pp. 101–112.

[5] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," *Accepted at IMC 2015*, 2015.

[6] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[7] K. Wiles, "The Pktgen Application," http://pktgen.readthedocs.org/en/latest/index.html, last visited 2016-2-24.

[8] N. Bonelli, "PFQ i/o," http://www.pfq.io/, last visited 2016-02-15.

[9] Ntop, "PF_RING," http://www.ntop.org/products/packet-capture/pf_ring/, last visited 2016-02-28.

[10] L. Rizzo, "The netmap project," http://info.iet.unipi.it/~luigi/netmap/, last visited 2015-11-04.

[11] P. Srivats, "Ostinato – packet traffic generator and analyzer," http://ostinato.org/, last visited 2016-02-27.

[12] L. Gorrie, "Snabb switch," https://github.com/SnabbCo/snabbswitch, last visited 2016-02-28.

[13] G. Luke, "Packet copies: Expensive or cheap?" https://github.com/snabbco/snabb/issues/648, last visited 2017-01-18.

[14] P. Emmerich, "libmoon," https://github.com/libmoon/libmoon, Technische Universität München.

[15] "Intel DPDK: Data Plane Development Kit," http://dpdk.org/, Intel Corporation, last visited: 2015-11-04.

[16] M. Pall, "The luajit project," http://luajit.org/, last visited 2016-2-29.

[17] P. Emmerich, "MoonGen," https://github.com/emmericp/MoonGen, Technische Universität München.

[18] "Open vswitcht," http://openvswitch.org/, last visited 2016-2-29.

[19] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet14)*, Luxembourg, oct 2014.