Architectures for Fast and Flexible Software Routers

Paul Emmerich, Sebastian Gallenmüller, Rainer Schönberger, Daniel Raumer, and Georg Carle
Technical University of Munich
Department of Informatics
Chair of Network Architectures and Services
{emmericp|gallenmu|schoenbr|raumer|carle}@in.tum.de

ABSTRACT

Network experiments contribute greatly to advancements in computer networks. Scientific prototypes being at the heart of this research should be able to reflect realistic conditions of current and future network environments. Only recently, specialized software frameworks such as DPDK or netmap have shown that handling 100 million packets per second is possible on affordable commodity server hardware. Building on this development, we designed MoonRoute, a flexible framework for building software routers with the ability to reuse existing components. MoonRoute also allows for quick prototyping of individual components in the packet processing pipeline of a software router with the Lua scripting language. Based on this framework, we discuss a proposed architecture for high-speed software routers. Our reference implementation of this architecture can saturate multiple 10 Gbit/s Ethernet ports with minimum-sized packets. These properties of our architecture make MoonRoute an attractive framework for conducting network experiments, investigating new components and applying them to network bandwidths of 40 Gbit/s and beyond.

MoonRoute, including the reference router implementation, runs on Linux and is available as free software under the MIT license.

CCS Concepts

•Networks → Network experimentation;

Keywords

Software routers; User space networking; Lua; DPDK

1. INTRODUCTION

Creating quick and dirty prototypes is a simple and effective way to demonstrate the feasibility of new ideas in scientific research. Network research in particular is driven by an almost dogmatic belief in running code [15]. Though, affordable small scale proof-of-concepts may have limited significance in a real world scenario. The situation is currently changing due to frameworks such as DPDK or netmap, which enable high performance software implementations on affordable hardware. We present MoonRoute, a framework to realize software routers achieving performance figures current implementations cannot provide [23, 40].

MoonRoute is an extensible framework for building software routers employing a high-performance architecture. We implement an architecture new to software routers by distributing packets between a simple, high-performance part and a slower, more sophisticated processing path. This architecture requires a new kind of batching called drop-out batching, to make use of this concept. The router comprises different modules glued together by the scripting language Lua. These modules can either be written in Lua – for rapid prototyping – or reuse existing C/C++ implementations for optimal performance. A foreign function interface conveniently allows code reuse in Lua. Performance is further increased by focusing on linear multi-core scaling, using immutable data structures and avoiding shared states.

The main contributions of this paper are:

- A novel architecture for software routers
- A new concept for batched packet processing
- An open source router experimentation framework

The paper begins with a short overview over related work in Section 2. We discuss the background in Section 3 explaining the approaches and frameworks for software routers. Design principles for architecture and approaches to batching are presented in Section 4; details of MoonRoute's implementation in Section 5. Benchmarking results, discussed in Section 6, demonstrate that MoonRoute is 40% faster than Click with batching and DPDK.

2. RELATED WORK

All major operating systems come with integrated routing capabilities which have been subject to benchmarking [11, 40, 23, 16, 49]. Alternative software routers have been proposed, the Click modular router being the most prominent one. On introduction Click achieved a fourfold increase in throughput over the Linux router while increasing flexibility at the same time [40]. The increased performance came from avoiding the overhead from interrupt handling in favor of polling, an optimization that has since been applied to the Linux router [3] and it thus has caught up. Though, Click remains an attractive framework in the scientific community for prototyping and receives many improvements, such as

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

DRAFT

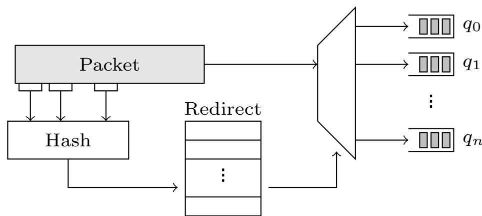

Figure 1: Receive side scaling

parallelization, batch processing, and IO optimizations [36, 13, 50, 51, 17, 39, 10].

In recent years, the performance of packet processing software has seen improvements by moving from the default network stacks to special-purpose packet IO engines such as netmap [50], PF_RING ZC [44], PacketShader IO engine (PSIO) [29], or DPDK [33]. Utilizing these IO engines, performance increases have been achieved for packet forwarding applications [50, 36, 29, 23, 26]. Once again Click is used as a benchmark of choice for netmap [50, 10], the PacketShader IO engine [29, 36], or DPDK [10], which achieved between a sixfold and a tenfold throughput improvement when using a specialized framework as back-end for Click.

Click has also been used in the area of network function virtualization (NFV): ClickOS implements service function chaining (SFC) with the help of click [42]. NFV with SFC is typically realized by connecting virtual machines, incurring a large overhead. Thus, frameworks moving networking functions from isolated VMs into containers or into embedded modules have been developed [52, 46]. MoonRoute can also be used as a NFV framework: we provide facilities to interconnect modules implementing a specific function. Our evaluation here is, however, focused on a router application.

3. BACKGROUND

MoonRoute and our reference router utilize several technologies and frameworks as building blocks. In the following we discuss important aspects of them.

High performance NICs.

Modern NICs offer an extensive feature set improving the performance of software routers. An important capability is the support for multiple queues that can be used independently from each other. For example, the Intel X540 we use features 128 receive and transmit queues [32], other NICs support up to 1536 [31]. Multiple transmit queues allow using the NIC from multiple threads simultaneously. Multiple receive queues are used to divide incoming traffic at the NIC. Synchronization overhead is eliminated by assigning CPU cores to queues for exclusive access. Incoming packets can be distributed to receive queues by hashing over packet headers; this is called receive side scaling (RSS) and illustrated in Figure 1. Explicit filters can also be configured, e.g., to direct packets handled by the control plane (e.g., ARP and ICMP) to a different queue.

CPU and memory architecture.

Modern server CPUs have a three-level cache architecture where the lower two levels are per-core exclusive (in the 100 kB range per core) and the third (around 8 to 20 MB) shared among all cores. NUMA (non-uniform memory-access) found in multi-CPU systems also needs to be considered: PCIe devices are local to a single CPU. For such systems accessing NICs from a remote CPU incurs an overhead.

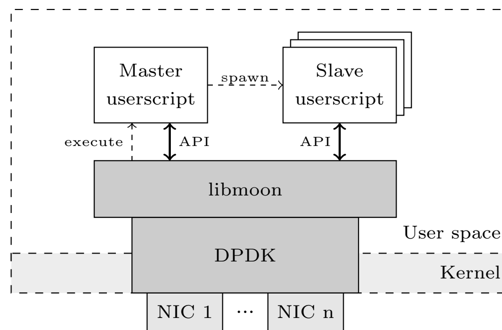

Figure 2: MoonGen architecture

DPDK.

The Data Plane Development Kit (DPDK) is an open source framework for high-speed packet IO by Intel [33]. It maps the whole PCIe memory space of a NIC into a user space process, thereby giving an application exclusive access to a NIC. DPDK comes with tightly integrated drivers for NICs from a wide range of vendors [19]. These drivers actively poll packets and the whole framework is heavily optimized for batch processing of packets. Moreover, data structures and memory allocators for packet buffers are simplified and optimized.In addition to its performance, DPDK provides extensive libraries of algorithms and data structures for packet processing. The wide range of supported NICs and the included libraries make DPDK an ideal backend for MoonRoute.

libmoon.

libmoon [21, 22] is a Lua wrapper for DPDK with multithreading based on multiple independent LuaJIT [45] VMs with utility functions for inter-VM communication. This architecture, shown in Figure 2, is suitable to build a multithreaded software router. The independent nature of the different threads is beneficial as we aim to avoid dependencies between threads to ensure linear scaling. The combination of Lua with DPDK allows easy and flexible composition of different router modules while still maintaining performance. Prototyping of modules with JIT compiled scripts is also feasible.

Design patterns for packet processing systems.

Network systems are often split into different tiers ranging from simple and fast forwarding to complex handling of high-level protocols. Routers can be split into two parts: The *data plane* handles the forwarding of packets, the *control plane* manages the routing table and implements high-level routing protocols [35]. This separation is clearly visible in high-end hardware routers where the two parts are often implemented on different hardware units within the same chassis, e.g., in Juniper [34] or Cisco [14] routers.

This separation also exists in software routers. For example, the whole kernel implementation of the Linux router

DRAFT

can be interpreted as the data plane of the router. The control plane can be left to a third-party implementation like BIRD [1], Quagga [2], or XORP [30] to provide the kernel with the routing table. However, this still leaves significant complexity like handling ICMP in the data plane – a task with lower priority that could be moved to the control plane to simplify the performance-critical part. Open vSwitch, a software OpenFlow switch, slims down the data plane to a bare minimum called the data path consisting only of a table lookup. A second separate user-space process implements OpenFlow and manages the low-level rules used by the data path [48]. These two components together represent the data plane of an OpenFlow system, the control plane is an external OpenFlow controller [43].

Simple functions are typically fast and complex functions slow. A good design clearly categorizes and separates functionality to achieve a high performance.

4. MOONROUTE ARCHITECTURE

RFC 1812 [8] defines the basic functional requirements for routers. We also focus on the following non-functional requirements.

- **High performance.** One server, using currently available hardware, should be able to saturate multiple 10 GbE ports with minimum-sized packets.
- **Multi-core scaling.** As the trend in processing hardware moves from multi-core to many-core architectures, the router should incorporate the benefits and challenges of this development.
- **Flexibility.** It should be possible to change and add functionality easily.
- **Code reuse.** Implemented components should be reusable with as few changes as possible.

In the following, we elaborate how these requirements are achieved in MoonRoute.

High performance & multi-core scaling.

Routers implemented with MoonRoute support multi-core scaling by respecting the following design principles:

1. Use lock-free message queues for synchronization.
2. Avoid working on shared data structures.
3. Use immutable data structures if sharing is necessary.

Incoming packets are distributed across multiple CPU cores at the NIC via the previously explained RSS feature, multiple TX queues are used to merge the outgoing packets. This offloads the challenging parts of packet flow parallelization and serialization to hardware. As routing information has to be shared among threads, we opted to provide an interface to integrate an immutable data structure.

Multi-CPU systems feature multiple PCIe hierarchies; each NIC has one local CPU. Other (remote) CPUs can only be accessed through expensive inter-CPU connections. For optimal efficiency, incoming packets should be handled by the local CPU to avoid unnecessary overheads.

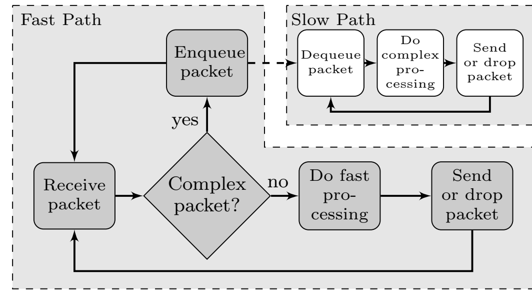

Figure 3: Lock-free handling of complex packets

Flexibility & code reuse.

Flexibility and the ability to reuse existing code is a main driver of software components in networking. The Click router [40] is popular because of its modular structure. Users customize software routers by changing the interconnects between predefined functional modules. However, adding new functionality to Click is only possible by writing new elements in C++.

We go one step further and implement the main loop in the scripting language Lua. This allows both changing the interconnect between pre-defined modules and simple introduction of new modules or algorithms. It also allows fast and flexible prototyping by changing or adding core functionality directly in Lua code as well as using modules optimized for performance, which may be implemented in C. The reuse of code is easily possible due to the foreign function interface of LuaJIT which allows embedding C into Lua. In the latter case, Lua code acts as glue, joining the modules together and defining the packet flow. The main forwarding loop of our reference router is less than 40 lines of code and consists of calls to predefined blocks and interconnect logic.

4.1 Separation into slow and fast components

Following the design patterns discussed in Section 3, MoonRoute splits the router into two main parts: Simple but efficient components for forwarding, i.e., validity checks, lookups in the routing table, and necessary packet modifications are implemented in the ***fast path***. Any packets requiring complex actions, e.g., ICMP responses or ARP lookups are forwarded to a ***slow path***. In the following, we refer to all directly routable packets as *simple packets*, to packets requiring additional processing (e.g., TTL expired) as *complex packets*. The slow path components also manage the routing table used by the fast path. This combination is roughly equivalent to what is offered by the router in the Linux kernel.

Figure 3 shows a control-flow graph of a packet through the fast and slow components. The slow path and fast path run in separate threads and communicate and pass packets via bounded lock-free queues. Note that the throughput of complex packets is not critical and the fast path will simply drop packets immediately if the communication queues overflow. In our implementation, the slow path is an order of magnitude slower than the fast path. But it is still capable of processing packets in the order of one million packets per second, which is sufficient to handle ICMP responses (which are typically rate-limited) and high-level protocols. The slow path is also responsible for a seamless connection to the OS

DRAFT

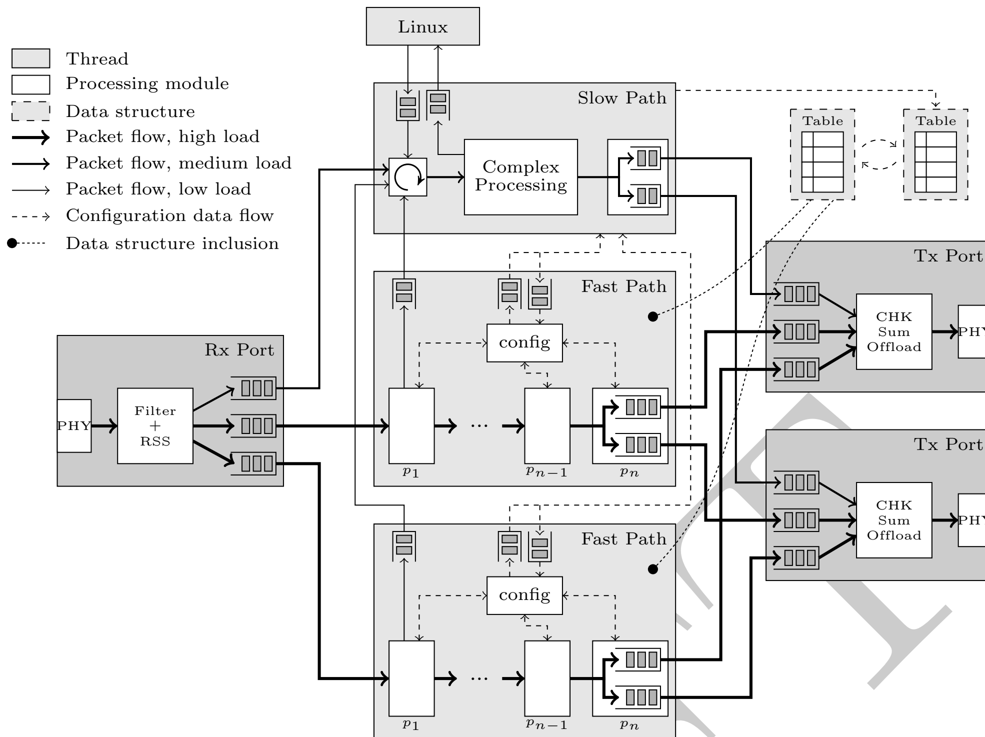

Figure 4: MoonRoute architecture

(omitted from the figure for simplicity). Packets that must be handled by the control plane, e.g., routing protocols, are forwarded to the OS and handled by a third-party control plane.

4.2 Architecture

MoonRoute contains three different types of components:

Fast path components perform fast packet forwarding according to forwarding rules specified in a routing table. Packet processing is done in batches using multiple processing steps $(p_1, p_2, ..., p_n)$.

Slow path component performs multiple tasks:

- Handles complex packets, which cannot be processed by the fast path.
- Performs synchronization for fast and slow path units (manage shared data).
- Provides a user interface.
- Exchanges information with the underlying OS to integrate third-party control planes.

Linux as underlying OS provides a general purpose network stack for advanced packet processing and protocols.

To allow scaling with an increasing number of CPU cores, multiple identical instances of the fast path are created. Each fast path runs in a separate thread pinned to a distinct CPU core. In the following, we define the number of ingress network ports as n_{rx} and the number of egress ports as n_{tx}. The number of fast path processing units shall be n_f, each fast path can read from multiple NICs via a round-robin arbiter. This configuration is read from a file on startup. Figure 4 shows an example of the router architecture with $n_{rx} = 1$, $n_{tx} = 2$ and $n_f = 2$.

Rx and tx ports.

On each rx port, incoming packets are distributed among all fast paths on the local CPU via hashing. Explicit hardware filters configured on the NIC direct packets addressed to the control plane to the slow path. Each tx port configures $n_f + 1$ tx queues. Note that all fast path units, regardless of NUMA association, need to access a tx port.

Fast path.

Each fast path unit reads packets from its dedicated rx queue on each rx port assigned to it in a round-robin fashion. Figure 4 omits arbitration as this example uses only one rx port to keep the figure simple. All processing steps p_i are user-replaceable modules. The last processing step p_n distributes the packets to the tx ports via a tx queue associated with this fast path.

There are two communication interfaces to the slow path via lock-free queues. The first escalates packets requiring complex but low-priority processing to the slow path. The second is a message-based synchronization interface to configure fast path units during runtime. For example, this is used for routing table updates.

Slow path.

The slow path manages the configuration and routing table and can be managed via a command line interface (CLI). It connects to all fast paths, all NICs, and the Linux kernel.

DRAFT

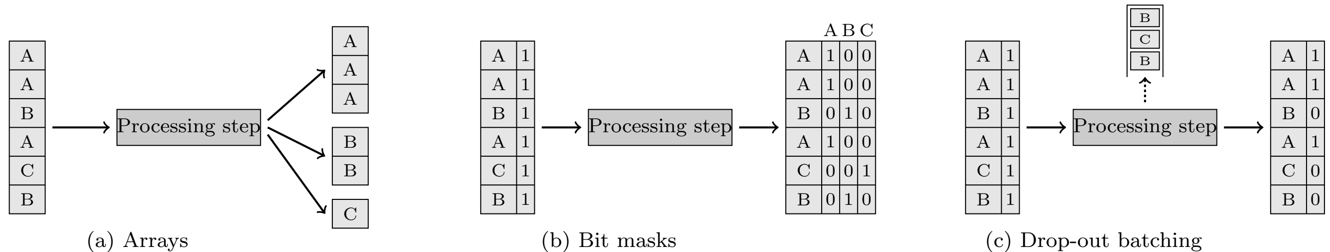

Figure 5: Batch scattering

Routing table.

To efficiently utilize the CPU caches, one routing table must be shared among all fast path instances. This represents a potential bottleneck for multi-core scaling. Therefore, we need a lock-free data structure. However, lookup data structures are often general-purpose implementations. Routing tables are special cases with a very large read/write ratio. One lookup is required per routed packet, but routers on the Internet update their routing table only a few hundred times per second [28]. These updates can be batched since a small latency is of little consequence.

We implement a *double buffering* approach and maintain two routing tables: an active and a passive one. The active table is immutable and used by the fast path units, the inactive table can be updated in the background by the slow path, cf. Figure 4. The slow path periodically initiates a table swap by sending a message to all fast paths which then swap the used pointer before the next batch of packets is processed. Once all fast paths have acknowledged the swap, the slow path begins updating the previously active table. This effectively immutable data structure in the fast path avoids cache contention (cf. Section 4). This operation can be implemented for any general-purpose data structure.

Similar solutions to multi-threaded scaling are used in other routing table implementations. For example, the Linux routing table uses the read-copy update (RCU) synchronization mechanism provided by the kernel [47]. Another example is the Poptrie data structure [7]. Both operate on a finer granularity and only swap modified parts of the data structure, we swap the whole table. Our approach is independent from the routing table implementation, which can be replaced. MoonRoute can be modified to support a natively multi-threaded data structure if such a data structure is to be benchmarked or tested.

4.3 Batch scattering

Batch processing is crucial to high-speed packet processing in software [36, 50, 26]. Batches improve cache performance as each processing step uses different memory regions and code, thus affecting both instruction and data caches.

Each batch is represented in a statically sized array that is passed to the processing units. However, different packets within a single batch may require different processing. For example, invalid packets must be dropped instead of being forwarded, complex packets must be sent to the slow path. Modules like firewalls can cause even more diverging paths within the fast path.

Two approaches for batch scattering to handle packet divergence found in the literature [36, 33] are: splitting batches into multiple batches on demand (Figure 5a) and using bit-masks to mask out packets for certain processing steps (Figure 5b). We introduce a hybrid between the two called *drop-out batching* (Figure 5a) that is optimized for forwarding a few packets to the slow path as shown in Figure 5c. Different batch scattering algorithms can be used depending on the processing steps. MoonRoute provides all three variants shown in Figure 5.

Arrays.

Figure 5a shows batch scattering used to implement batch processing in Click [36]. One processing step outputs multiple arrays that are passed to the following processing steps. A disadvantage shows if only few packets diverge from the main processing path. This affects both the diverging packets and the normally processed packets. The former are now in a small batch which affects performance, the latter need to be moved just to remove a few packets from the original batch. Smaller batches can be *rebatched*, i.e., buffered and combined with the next batch of packets. This avoids smaller batches at the cost of additional complexity and overhead.

Bit masks.

A second approach is used in multiple modules of the DPDK packet processing library [33]. Bit masks mask packets for processing steps as shown in Figure 5b. Processing steps accept bit masks as additional inputs and outputs. These bit masks are only as large as the maximum batch size, i.e., 64 bit to 128 bit in a typical configuration. Only packets with the corresponding bit set are processed by the module, all other packets are ignored. Outputs of processing steps mark packets in different bit masks. For example, a module implementing RFC 1812 [8] validity checks can mask out invalid packets and free the associated memory. Subsequent processing steps then ignore the invalid packets in the batch completely and the packets are implicitly dropped at the end as the output step ignores them as well.

This is based on the assumption that only few packets are masked out, which is the case in the validity check example. This approach still performs some unnecessary work for masked out packets in the fast path. The total performance is not impacted and flooding the router with packets that must be masked out is not a threat. The router needs to be able to handle the same amount of valid packets. An invalid packet requires less processing time than a valid packet in the fast path.

Drop-out batching.

We implement *drop-out batching*, a hybrid approach to handle small path divergences, e.g., handling complex pack-

DRAFT

ets. As visualized in Figure 5c, the processing step also operates on a bit mask and masks out packets, but it additionally gathers marked packets in a separate new batch.

Our reference router uses this in all processing steps except the last: Distributing packets to different NICs is done with batch scattering and rebatching, see Figure 6. The decision for this technique is based on the assumptions that simple packets requiring high processing performance (‘A’ in Figure 5c) constitute the biggest share and traverse the same processing path; complex packets, requiring different processing (‘B’ & ‘C’ in Figure 5c) may be treated with low priority or dropped.

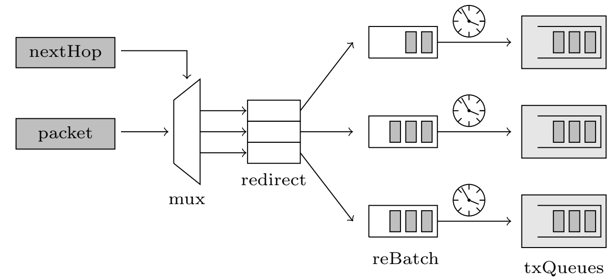

Figure 6: Packet distribution with rebatching

5. ROUTER IMPLEMENTATION

In this section, details about the MoonRoute reference implementation and its building blocks are explained.

5.1 Packet processing modules

Lua code connects different processing modules. The modules are implemented in a mixture of C and Lua code and provide a documented interface for use in the Lua programming language. For most of the modules multiple implementations are available.

5.1.1 *IP validity checking*

MoonRoute performs an IP header validation described by RFC 1812 [8]. It checks the length reported by the link layer and the IP header length for the minimum value of 20, the IP version, and the IP checksum. The last three tests are already performed in hardware, so these results can be read from the metadata of a packet. For simplicity, the total length of the IP packet is not checked against the actual packet length but merely if it is large enough to contain a minimum sized header of 20 B.

5.1.2 *Longest Prefix Matching*

This module performs Classless Inter-Domain Routing as described in RFC 4632 [25]. Our module is based on the longest prefix matching (LPM) library in DPDK [20]. The LPM module is a variation of the DIR-24-8 [28, 27] algorithm which is based on a trade-off between lookup speed and memory usage.

Our configuration allows up to 256 next hop entries resulting in a routing table size of 33.6 MB. A sufficient number according to Asai et al. who found a median of 125 next hops and an average of 219.5 ($\sigma = 202.9$) on $n = 39$ routers [7] in December 2014.

Unfortunately, the DIR-24-8 algorithm does not scale to IPv6. Our module is meant as a prototype and can easily be replaced to test and benchmark other algorithms under realistic conditions. A prime candidate would be the Poptrie [7] data structure. However, we include the DPDK LPM library by default as the license for the published source code of Poptrie forbids commercial use [6] and is thus incompatible with MoonRoute’s MIT license.

5.1.3 *Route application*

Looking up the route provides the required information but does not modify the packet for forwarding. We perform this step in a separate module for two reasons: flexibility and performance. Maintaining flexibility, the contents of the `nextHop` entry in the routing module are defined by the route application module and opaque to the routing module. Our default module stores the MAC address of the next hop and the associated interface. However, a more complex module can store more information, e.g., to support equal cost multi-path routing, without modifying the lookup table. The second reason is increased performance due to better cache-locality when processing batches. The lookup operates on the lookup table which stores only indices into the next hop table. Since the information from the next hop table is not yet required during lookup, it does not have to be loaded into the cache while the destinations for a whole batch are looked up. Thus, this two-step process minimizes the overall cache pressure.

5.1.4 *TTL decrement*

This module updates the TTL field in the IP header according to RFC 1812 [8]. Expired packets are forwarded to the slow path for further processing.

5.1.5 *Packet distribution*

The final step sends out packets and causes the largest diversion in paths taken by packets as different packets in the same batch can be sent to different NICs. As shown in Figure 6, this module implements a packet multiplexer controlled by the `nextHop` entry. Naïvely splitting packets at transmit time and immediately transmitting them results in small transmit batches when using multiple NICs. Batching at transmit time is important since the call into the driver incurs an overhead independent of the batch size. Therefore, we implemented *rebatching*, i.e., one queue per NIC in the distributor module to accumulate packets to send out larger batches at once. Each queue has a configurable size and timeout to avoid starving and excessive latencies at low loads. Queue management algorithms like RED [12] can also be embedded in this step.

5.1.6 *Packet filters*

Three different filters are implemented for packet classification and can be used to implement firewall rules or to forward certain packets directly to the slow path. The first two filters are hardware filters for Intel NICs. One matches layer 2 packets, e.g., to forward ARP packets directly to the slow path. The second is a 5-tuple filter matching on layer 3 addresses and the layer 4 protocol and ports. This filter is used to efficiently handle traffic addressed to the control plane. The third filter is the DPDK ACL library [18] that also implements 5-tuple filtering. It uses bit mask based batch scattering for differentiated processing in the fast path. Hardware filters can only choose an rx queue and thus a specific fast path or the slow path.

DRAFT

5.2 Fast Path

The source code of the fast path main loop connecting the submodules consists of only about 100 lines of Lua code and is meant to be modified to customize MoonRoute. Packet reception is handled via a round-robin arbiter, i.e., a loop over all queues containing the packet processing logic. The main component of the fast path is the interconnect between the previously discussed modules, i.e., glue logic written in Lua. It initializes several bit masks and calls the processing steps; the reference router includes six such steps. The last component are periodic tasks: flushing rebatching queues in the packet distributor and handling commands from the slow path.

5.3 Slow Path

The slow path handles complex packet processing and additional low priority tasks in the same thread. Currently the slow path is single-threaded since it only handles tasks with a low priority or low throughput. By moving synchronization and user interface tasks into a separate thread, the functionality can be parallelized.

5.3.1 Routing table management

As mentioned in Section 4.2, the routing table is *double buffered* to allow lock free-operation. The slow path updates and synchronizes the tables from three different sources: *Static routes* read from the router configuration file, *dynamic routes* configured during runtime via a CLI, and *Linux routes* extracted from the Linux host system.

Updating the routing table works as follows. Routing entries are retrieved from these sources and the next hop IP addresses are resolved to MAC addresses via the ARP table that is also in the slow path. Our double buffered approach to routing tables ensures that the slow path has exclusive access to a copy of the routing table to modify it. Finally, a pointer to the new routing table is sent to all fast path units via the command queues. Fast paths confirm that they are using the new table via a message back to the slow path.

5.3.2 Packet processing

The slow path receives and processes packets from three sources: ARP packets arriving at the NIC ports are redirected to the slow path where they are handled to maintain an ARP table and to respond to ARP requests. Fast paths deliver complex packets to the slow path for further processing. Finally, the slow path provides Linux network interfaces through which packets can be exchanged.

The slow path implements a full router but is less optimized for performance. Packets generated by the slow path, e.g., ICMP responses, are handled by this routing process directly in the slow path without reentering the fast path. This keeps the fast path as slim as possible.

Flows to local subnets (subnets with no next hop IP that are directly reachable) also need special attention: the fast path needs to know the MAC addresses of each directly reachable host. It usually gets the required MAC address from the routing table which directly stores the MAC address to avoid an unnecessary ARP lookup. If the destination IP of a local subnet is unknown the packet is sent to the slow path which performs the ARP lookup. A new dynamic /32 route to the destination IP is created by the slow path. Hence, subsequent packets to the same destination can be handled by the fast path.

5.4 Interaction with the Linux network stack

MoonRoute is transparent to router control planes and can exchange packets with the host stack using DPDK's Kernel Network Interface (KNI) library. NIC ports bound to DPDK are invisible to the Linux kernel, they are exclusively being used by DPDK and the kernel is not aware of them. Therefore, we create a virtual twin with the same MAC for each physical port. All egress traffic from Linux to these virtual ports is forwarded to their physical counterpart. Ingress ARP packets are cloned and forwarded to fill the Linux ARP table. Incoming packets are also sent to the virtual port if the interface itself is addressed

Advanced routing protocols implemented in BIRD [1], Quagga [2], or XORP [30] running on the Linux host can be made available to MoonRoute. They see all necessary incoming messages from the virtual twin interfaces and can send messages through them. Control planes built on Linux leave the actual routing to the Linux kernel by setting routes in the kernel routing table. This routing table is periodically copied by MoonRoute.

6. EVALUATION

MoonRoute outperforms commonly used software routers, we show this through benchmarking in this section.

6.1 Methodology

All tests use the following setup and configuration.

Hardware.

We use two different servers as device under test (DuT) to evaluate MoonRoute: one representing a low-end server and one a mid-range server for heavier workloads. The first server features a quad-core 3.2 GHz Xeon E3-1230 CPU with 8 MB L3 cache and a dual 10 Gbit/s Intel X520-T2 NIC, the second an octa-core 2 GHz Xeon E5-2640 v2 CPU with 20 MB L3 cache and two dual 10 Gbit/s Intel X540-T2 NICs. Hyper-threading is disabled on both CPUs to avoid unintentional resource contention.

The CPUs' clock frequencies are set manually with power-saving features disabled. Manually reducing the frequency allows us to artificially enforce a CPU bottleneck in scenarios where the throughput would usually be limited by other factors.

Software.

All servers were running Debian with a Linux 3.7 kernel. `perf` [41] together with `pmu-tools` [37] was used to measure cache behavior. The LuaJIT profiler was used to measure relative CPU usage of the different modules.

MoonGen was used as packet generator and counter on a separate server with direct connections to the DuT. Latency measurements where conducted using MoonGen's hardware timestamping [22].

Configuration and test traffic.

The following parameters were used for each experiment unless mentioned otherwise.

- Constant bit rate (CBR) test traffic with minimum-sized packets (64 B)
- Traffic was randomly split between used output ports
- Rx and tx batch sizes of 128

DRAFT

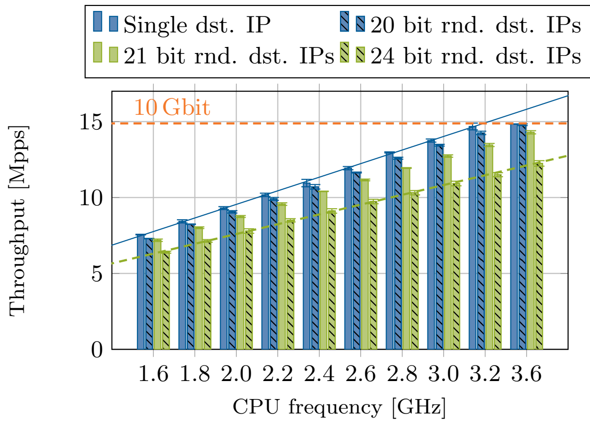

Figure 7: Scaling with the CPU frequency

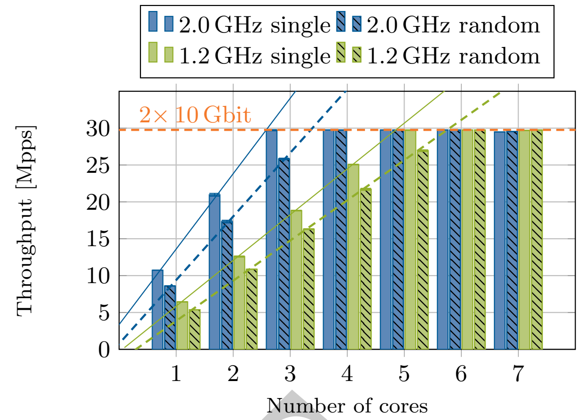

Figure 8: Scaling with the number of CPU cores

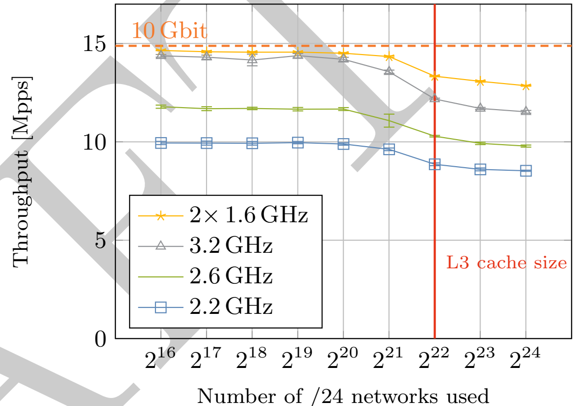

Figure 9: Effects of the routing table on throughput

- Latency measurements are based on at least 40 000 random samples

The throughput of DPDK-based applications is very stable within a single test run consisting of an uninterrupted stream of packets. However, it varies by up to 2% between different runs. This effect is not specific to MoonRoute but an effect of the `ixgbe` driver in DPDK. We could reproduce this with a simple DPDK application that only sends out packets without any processing on `ixgbe`-based NICs. The effect disappears when using a NIC with the `i40e` driver. We repeat all throughput tests four times and report the average and standard deviation (typically less than 1% relative deviation).

6.2 Single core forwarding throughput

Forwarding traffic from one port back to the same port using only a single fast path thread gives a baseline performance in Figure 7. We vary the CPU frequency and apply a linear regression to the results, demonstrating linear scaling (results limited by the 10 Gbit/s line rate at 3.6 GHz were excluded).

We also randomize the varying ranges of the upper 24 bits of the destination IP address to get a baseline for routing table lookup performance. Note that the contents of the routing table do not matter for the performance in this scenario: the DIR-24-8 algorithm uses the upper 24 bits as index and the single large default route we installed is realized as 2^{24} rules in the data structure [28].

6.3 Packet sizes

We tested packet sizes between 64 B and 544 B on a CPU underclocked to 1.2 GHz. The size does not influence the throughput measured in packets per second unless limited by the line rate of the network link.

6.4 Multi-core scaling

We use our octa-core server to show that MoonRoute scales beyond one thread by forwarding traffic with two 10 Gbit/s ports with an increasing number of fast-path units. Figure 8 plots how MoonRoute achieves perfect multi-core scaling independent of the routing table utilization and clock frequency.

Figure 9 shows the behavior with an increasing number of used routing table entries. MoonRoute benefits more from an increased number of cores than from faster cores: The throughput of two cores clocked at 1.6 GHz is higher than one core at 3.2 GHz. This can be explained by looking at caches. Adding an additional core adds not only processing power but also additional cache capacity (256 KiB on this CPU). This result shows that MoonRoute is well adapted for multi-core architectures. The following results are limited to a single core for simplicity.

6.5 Routing table performance

Overloading the L3 cache leads to the performance degradation as depicted in Figure 9. We measure the number of cache misses to show that this is the bottleneck. Figure 10 shows the throughput and cache misses of single-core forwarding at 3.2 GHz. The correlation between the drop in achieved throughput and the increase in L3 cache misses demonstrates that the cache size is the bottleneck.

6.6 Batching

MoonRoute uses both tx and rx batching, both affect the throughput and latency. We underclocked the CPU frequency slightly to 3.0 GHz and used random traffic with 2^{24} routing table entries. The lower speed avoids coming close to line rate, the large routing table presents a worst-case scenario for the caches.

DRAFT

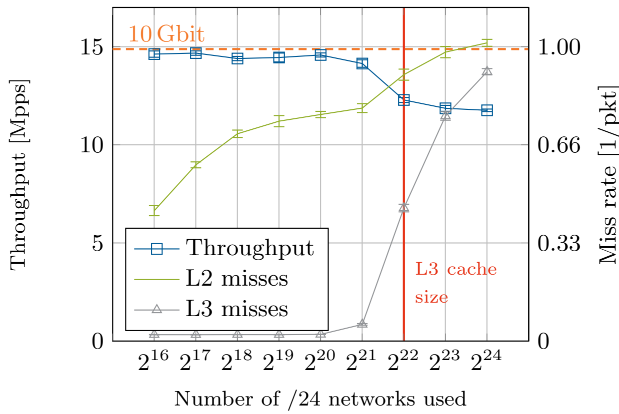

Figure 10: Cache effects (3.2 GHz)

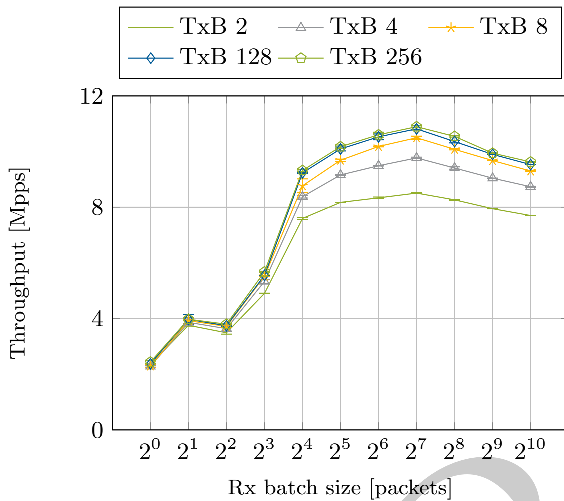

Figure 11: Effects of batching (3.0GHz)

Throughput.
Figure 11 shows the throughput achieved by different rx and tx batch configurations. Increasing the batch sizes leads to a higher throughput. The optimum throughput for MoonRoute is reached at batch sizes of 128, increasing either rx or tx batch sizes further lowers the performance. The impact of tx batching is smaller than rx batching: only one processing step is affected by the tx batch size, while rx batching affects all.

Profiling.
We look at the CPU's hardware profiling counters to explain the observed effects. Figure 12 shows the cache and branch misses on average per packet as a function of the rx batch size. The tx batch size is fixed to 128. As mentioned above, we use random destination addresses here. Hence, the large base-line for cache misses (cf. Figure 10). Low performance at low batch sizes correlates with a high branch miss-prediction rate that virtually disappears at batch sizes of 16 and above. It remains unclear from this experiment what causes the increase of 20% when going from batch size 16 to 128. Higher batch sizes suffer due to contention in the L1 cache.

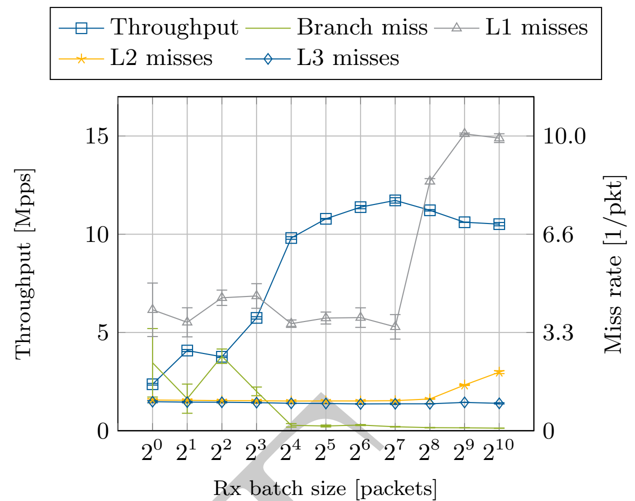

Figure 12: Hardware events (3.2 GHz)

Latency.
This increased performance at larger batch sizes comes at the cost of latency. Measuring latency correctly is a challenging task. One cannot simply apply full line rate as this would fill up all buffers – resulting in an unrealistic worst-case latency. Neither can a low constant rate be applied as the packet rate determines the fill speed of batches and thus affects latency. We therefore applied 90% of the previously determined maximum throughput for each tested configuration as test traffic for the latency tests.

Figure 13 shows the latency under increasing rx batch sizes. We only include results for tx batch sizes 128 (optimum performance) and 8 (only 3% slower than the optimum, cf. Figure 11) to keep the graph readable. Large rx and tx batch sizes or a large tx to rx batch size ratio increases latency. Considering both the throughput in Figure 11 and the latency here, the optimum rx batch size is between 16 and 64. Tx batch size of 8 can be preferred for latency-sensitive applications. Note the effective tx batch size for latency considerations depends on the number of tx ports (1 in this scenario) and a low timeout for the tx buffer should be configured if many ports are used (cf. Section 5.1.5).

The rx batch size represents only an upper bound for the batch size: the input module simply retrieves all available packets from the NIC. A batch that is not completely full means that there is spare processing capacity and continuing with the smaller batch does not affect throughput. This leads to the outlier and comparatively high latency in Figure 13 for rx and tx batch size 128. MoonRoute runs at 90% capacity, resulting in smaller rx batches and some packets spend time in the tx rebatching process.

6.7 Code profiling

We use the profiler included in LuaJIT to estimate the time spent in the different modules. This is possible since the top-level module is written in Lua and all modules start with a Lua function that calls into optimized C versions.

Table 1 shows the time spent in the different modules when forwarding random packets. A faster routing table module based on a modern data structure can increase the performance significantly: 33% of the CPU time is spent in

DRAFT

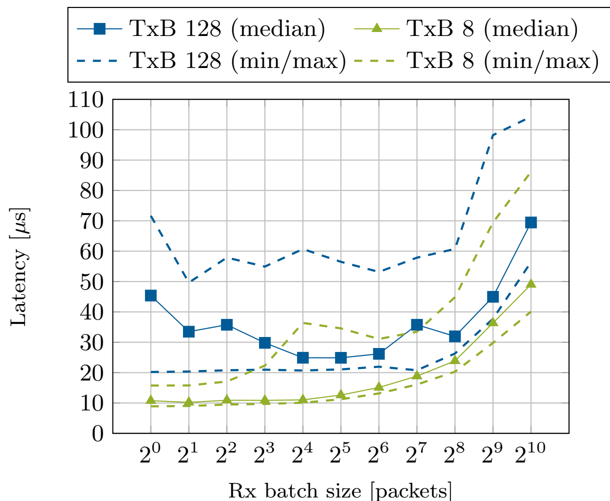

Figure 13: Latency (90% of maximum throughput)

Module	Relative CPU usage
Distribute and send	35%
Routing	24%
Receive	17%
Apply route	9%
Other*	16%

* Includes glue logic and checksum update.

Table 1: Time spent in modules

the routing table and route application module.

6.8 Comparison with DPDK Click

We run the single-core forwarding test described in Section 6.2 on the Linux router, vanilla Click [40] with DPDK enabled, and on FastClick [10] with both DPDK and batching enabled on the same hardware. Table 2 shows the resulting throughput: MoonRoute is about 4 Mpps (40%) faster than FastClick. Expanding the test to two cores resulted in the same performance for FastClick and MoonRoute: 14.88 Mpps.

We used the fastest routing table available in Click: `RangeIPLookup` [24]. Inserting into this table during run-time resulted in a significant impact on the forwarding performance: 99% of the packets were dropped during large modifications via the `ctrl` handler of the socket API. Inserting a large number of rules was also too slow for tests with 2^{24} routing tables entries in Click. MoonRoute's architecture with double-buffering for routing tables and the clear distinction between control and forwarding plane prevents such problems. The configuration files and packet generator script used by us are available in our git repository [4].

6.9 Comparison with published performance data

Table 3 shows a comparison with other available software routers. The table consists of our own measurements from Table 2 and published measurements scaled to be comparable to MoonRoute. We assume that the other routers can also scale linearly with the CPU's frequency and the number

Implementation	Routing tbl.	Mpps	Relative
MoonRoute	1	14.6	100%
MoonRoute	2^{20}	14.2	97%
MoonRoute	2^{24}	11.6	79%
FastClick DPDK*	1	10.4	72%
FastClick DPDK*	2^{20}	10.4	72%
Click DPDK†	1	4.3	29%
Click DPDK†	2^{20}	4.2	28%
Linux 3.7	1	1.5	11%

* Batching, git revision 57177d2 from [9]
† No batching, git revision 2c6c837 from [38]

Table 2: Maximum single core forwarding performance comparison.

of cores. Note that older publications have a slight disadvantage in this comparison due to improvements in CPU architecture (the 3.2 GHz CPU used by us was released in 2011). Despite these limitations, the table gives a rough overview over the performance compared to other software routers.

Of particular interest are the results of the 6WIND Turbo Router that achieves 75% of MoonRoute's throughput while claiming production quality. It is unfortunately proprietary and was not available for evaluation on our hardware.

7. CONCLUSION

MoonRoute provides a flexible prototyping and experimentation platform for building software routers. It features a seamless integration of user-provided Lua scripts allowing quick prototyping of new modules and features. Modularity, flexibility and high performance – often considered as conflicting optimization goals – are achieved by our architecture through careful design choices: the separation of performance critical tasks from low priority tasks, the application of new batching techniques to optimize CPU utilization, and an architectural focus on multi-core scalability. The performance evaluation of our reference router offers insights in its scalability, the positive effects of batching, and influence of cache effects. The overall performance achieved by our architecture in MoonRoute is superior to similar frameworks and implementations. Moreover, latency measurements for routing are included, a dimension often neglected by other publications.

Limitations.

MoonRoute is not meant for production networks, only to prototype specific features in a data plane. We do not implement all edge cases: most notably, MoonRoute currently does not support fragmentation of packets, IP extension headers and multicast, i.e., MoonRoute is not fully compliant with RFC 1812. However, our design ensures that edge cases like fragmentation can be handled in a flexible way without impacting the performance of the core forwarding engine.

NUMA support is still preliminary and requires manual mapping of threads to CPU cores for optimum performance. Only a single slow path unit is supported; NUMA setups might benefit from multiple slow paths.

DRAFT

Implementation	Source	Routing tbl.	CPU freq.	Mpps	Mpps scaled to 3.2 GHz	Relative
MoonRoute	—	1	3.2 GHz	14.6	14.6	100%
MoonRoute	—	2^{20}	3.2 GHz	14.2	14.2	97%
MoonRoute	—	2^{24}	3.2 GHz	11.6	11.6	79%
6WIND Turbo Router	[5]	Unknown	2.8 GHz	9.6	11.0	75%
FastClick (DPDK 2.2)*	–	1	3.2 GHz	10.4	10.4	72%
FastClick (DPDK 2.2)*	–	2^{20}	3.2 GHz	10.4	10.4	70%
Batching Click (PSIO)	[36]	Unknown	8x 2.66 GHz	41.7	6.27	43%
Click (DPDK 2.2)†	–	1	3.2 GHz	4.32	4.32	29%
Click (DPDK 2.2)†	–	2^{20}	3.2 GHz	4.18	4.18	28%
FreeBSD 11-routing	[16]	2	8x 2.0 GHz	9.5	1.91	13%
Route Bricks	[17]	2^{18}	8x 2.8 GHz	12	1.71	12%
Linux 3.7	–	1	3.2 GHz	1.5	1.5	10%
Click	[40]	8	0.7 GHz	0.35	1.60	11%
FreeBSD 10.2	[49]	8	4x 2.13 GHz	1.78	0.67	4.6%
Linux 2.2.14	[40]	8	0.7 GHz	0.095	0.43	2.9%

* Batching, git revision 57177d2 from [9]
† No batching, git revision 2c6c837 from [38]

Table 3: Maximum single core forwarding performance comparison.

Reproducible research.
A development snapshot of our internal repository is publicly available on GitHub at [4], this version was used for the measurements in this paper. The repository also contains scripts and configuration files used to perform the evaluation, the raw data for the plots, and the Click configuration file used for the comparison.

Acknowledgments

This research is supported by the German BMBF projects DecADe (16KIS0538) and SENDATE-PLANETS (16KIS0472) The authors alone are responsible for the content of the paper.

DRAFT

8. REFERENCES

[1] BIRD. http://bird.network.cz/. Accessed: 2017-01-21.
[2] Quagga. http://www.nongnu.org/quagga/. Accessed: 2017-01-21.
[3] *ALS '01: Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, Berkeley, CA, USA, 2001.
[4] MoonRoute development snapshot, scripts, raw data, and click configurations. https://github.com/emmericp/MoonRoute-data, 2017.
[5] 6WIND. 6WIND Turbo Router. http://www.6wind.com/products/6wind-turbo-router/. Accessed: 2017-01-21.
[6] H. Asai. An Implementation of Poptrie IP Routing Table Lookup Algorithm. https://github.com/pixos/poptrie, 2016. Accessed: 2017-01-21.
[7] H. Asai and Y. Ohara. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 57–70. ACM, 2015.
[8] F. Baker. Requirements for IP Version 4 Routers. RFC 1812, RFC Editor, June 1995.
[9] T. Barbette. FastClick - A faster version the Click Modular Router. https://github.com/tbarbette/fastclick, 2016. Accessed: 2017-01-21.
[10] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *Architectures for Networking and Communications Systems (ANCS)*, pages 5–16. IEEE, 2015.
[11] R. Bolla and R. Bruschi. Linux Software Router: Data Plane Optimization and Performance Evaluation. *Journal of Networks*, 2(3):6–17, June 2007.
[12] B. Braden, D. D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and

L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309, RFC Editor, April 1998.

[13] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *USENIX Annual Technical Conference, General Track*, pages 333–346, 2001.

[14] Cisco. Cisco ASR 1000 Series Aggregation Services Routers. http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/white_paper_c11-452157.pdf, 2015. Accessed: 2017-01-21.

[15] D. Clark. A Cloudy Crystal Ball - Visions of the Future. In *Proceedings of the Twenty-Fourth Internet Engineering Task Force*, pages 539–543, July 1992.

[16] O. Cochard-Labbé. fbsd11-routing.r287531. https://github.com/ocochard/netbenches/tree/master/Xeon_E5-2650-8Cores-Chelsio_T540-CR/fastforwarding-pf-ipfw/results/fbsd11-routing.r287531, 2015. Accessed: 2017-01-21.

[17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28. ACM, 2009.

[18] DPDK. ACL Library. http://dpdk.org/doc/guides/prog_guide/packet_classif_access_ctrl.html. Accessed: 2017-01-21.

[19] DPDK. Supported NICs. http://dpdk.org/doc/nics. Accessed: 2017-01-21.

[20] DPDK. LPM Library. http://dpdk.org/doc/guides/prog_guide/lpm_lib.html, 2015. Accessed: 2017-01-21.

[21] P. Emmerich. libmoon. https://github.com/libmoon/libmoon, 2017.

[22] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[23] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems. In *Fourteenth International Conference on Networks (ICN 2015)*, Barcelona, Spain, Apr. 2015.

[24] E. K. et al. Click IPRouteTable Element Documentation. http://www.read.cs.ucla.edu/click/elements/iproutetable, 2016. Accessed: 2017-01-21.

[25] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. BCP 122, RFC Editor, August 2006.

[26] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, G. Carle, et al. Comparison of frameworks for high-performance packet IO. In *Architectures for Networking and Communications Systems (ANCS)*, pages 29–38. IEEE, 2015.

[27] P. Gupta. *Algorithms for routing lookups and packet classification.* PhD thesis, Stanford University, 2000.

[28] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1240–1247. IEEE, 1998.

[29] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[30] M. Handley, O. Hodson, and E. Kohler. XORP: An open platform for network research. *ACM SIGCOMM Computer Communication Review*, 33(1):53–57, 2003.

[31] Intel. *Intel Ethernet Controller XL710 Datasheet*, 2014. Rev. 2.1.

[32] Intel. *Intel Ethernet Controller X540 Datasheet*, 2015. Rev. 2.8.

[33] DPDK. http://dpdk.org/. Accessed: 2017-01-21.

[34] Juniper. Understanding MX Fabric. http://kb.juniper.net/InfoCenter/index?page=content&id=KB23065&actp=search, Sept. 2012. Accessed: 2017-01-19.

[35] H. Khosravi and T. Anderson. Requirements for Separation of IP Control and Forwarding. RFC 3654, RFC Editor, November 2003.

[36] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 14. ACM, 2012.

[37] A. Kleen. Intel PMU profiling tools. https://github.com/andikleen/pmu-tools/tree/d70840ba, 2015. Accessed: 2017-01-21.

[38] E. Kohler. The Click modular router: fast modular packet processing and analysis. https://github.com/kohler/click, 2016. Accessed: 2017-01-21.

[39] E. Kohler, B. Chen, M. F. Kaashoek, R. Morris, and M. Poletto. Programming language techniques for modular router configurations. Technical report, Technical Report MIT-LCS-TR-812, MIT Laboratory for Computer Science, 2000.

[40] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[41] Linux. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php?title=Main_Page&oldid=3535, 2015. Accessed: 2017-01-21.

[42] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr. 2014. USENIX Association.

[43] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[44] ntop. PF_RING ZC (Zero Copy). http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/. Accessed:

DRAFT

2017-01-21.

[45] M. Pall. LuaJIT. http://luajit.org/. Accessed: 2017-01-21.

[46] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, GA, 2016. USENIX Association.

[47] M. Paul. What is RCU, Fundamentally? https://lwn.net/Articles/262464/, 2007. Accessed: 2017-01-21.

[48] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[49] B. R. Project. Forwarding performance lab of an IBM System x3550 M3 with 10-Gigabit Intel 82599EB. http://bsdrp.net/documentation/examples/forwarding_performance_lab_of_an_ibm_system_x3550_m3_with_10-gigabit_intel_82599eb, 2015. Accessed: 2017-01-21.

[50] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[51] W. Sun and R. Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 25–36. IEEE Press, 2013.

[52] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 3–17, New York, NY, USA, 2016. ACM.

DRAFT