SUPERFLUIDITY

A SUPER-FLUID, CLOUD-NATIVE, CONVERGED EDGE SYSTEM

Research and Innovation Action GA 671566

DELIVERABLE I5.1:
FIRST REPORT ON FUNCTION ALLOCATION ALGORITHMS IMPLEMENTATION AND EVALUATION

Deliverable Type:	Report
Dissemination Level:	PU
Contractual Date of Delivery to the EU:	M11 (31/05/2016)
Actual Date of Delivery to the EU:	M11 (31/05/2016)
Workpackage Contributing to the Deliverable:	WP5

Editor(s):	ULG
Author(s):	**ULG (Cyril Soldani, Laurent Mathy), Intel (Vincenzo Riccobene, Michael Mcgrath), OnApp (Michal Belczyk, Michail Flouris, Anastassios Nanos, Julian Chesterfield, John Thomson)**
Internal Reviewer(s)	

Abstract:	
Keyword List:	

INDEX

List of Figures

List of Tables

Glossary

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
API	Application Programming Interface
ARP	Address Resolution Protocol
ATA	AT (Advanced Technology) Attachment
BF	Bloom Filter
BSD	Berkeley Software Distribution
CLT	Coron Lepoint Tibouchi
CPU	Central Processing Unit
DMA	Direct Memory Access
DPDK	Intel's Data Plane Development Kit
FIB	Forward Information Base
FIFO	First-In First-Out
GPL	GNU General Public License
GPU	Graphics Processing Unit
GSO	Generic Segmentation Offload
I/O	Input / Output
ICMP	Internet Control Message Protocol
IOV	I/O Virtualization
IP	Internet Protocol
IRQ	Interrupt ReQuest
KPI	Key Performance Indicator
MB	MiddleBox or MegaByte
MQ	Multi-Queue
NAPI	New API (see note 2 on page 19)
NAT	Network Address Translation
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NIC	Network Interface Controller
NUMA	Non-Uniform Memory Access

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
PCAP	Packet CAPture
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PEKS	Public-key Encryption with Keyword Search
PNFV	Private NFV
PSIO	PacketShader I/O
RAM	Random Access Memory
RAN	Radio Access Network
RSS	Receiver-Side Scaling
RX	Receive
SDN	Software Defined Network
SFP	Small Form-factor Pluggable
SLA	Service Level Agreement
SR-IOV	Single-Root IOV
TCP	Transmission Control Protocol
TSO	TCP Segmentation Offloading
TTL	Time-To-Live field of IP header
TX	Transmit
UDP	User Datagram Protocol
vCPU	Virtual CPU
VM	Virtual Machine
ZC	Zero Copy

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

1.1 Deliverable Rationale

1.2 Quality Review

Review Team member responsible of the deliverable: _____

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE

1.3 Executive summary

1.3.1 Deliverable description

This deliverable is an intermediate advancement report about the work-in-progress on task 5.1, *Function allocation algorithms implementation and evaluation*.

This task will leverage the block and functional abstractions developed in Tasks 4.2 and 4.3 to match a set of network services (decomposed into functional abstractions) to the underlying, available block hardware abstractions. This task will identify SLA (Service Level Agreement) metrics (e.g. throughput, delay, power consumption, etc.) and manage their mapping onto platform sources, etc). The problem solved by this task is thus the optimization problem of minimizing the platform resource usage while meeting individual SLAs, in order to automatically derive the best performance out of a platform. The work will include implementing algorithms that solve the allocation problem and evaluating for efficiency (e.g., in terms of goodput, delay, packet loss and number of services) when asked to install different sets of network services. The algorithms will further include dynamic re-evaluation of allocations in real-


time, e.g., in order to adapt to changing traffic patterns (e.g., to pin virtual machines to different CPU cores in order to better cope with a traffic spike).

For instance, we will focus on dynamic allocation and scheduling techniques for data-flow task graphs representing protocol stack application. For this purpose, we will make use of baseband protocol models and hardware abstractions developed in WP4. Task-centric modeling of baseband protocol processing functions allows exploiting inherent parallelism and heterogeneity of application for workload distribution to processor/thread pool resulting in accelerated execution of the application. Currently, the static methods are usually preferred for task mapping due to real-time guarantee, however, these suffers from inefficient use of resources resulting in system over-provisioning. We will study dynamic scheduling techniques enabling task distribution of multiple baseband stacks to underlying heterogeneous hardware resources for purpose of Cloud-RAN applications and develop the methods (e.g. hierarchical scheduling) that will enable to cope with scalability issues associated with modularized hardware platform.

1.3.2 Summary of results

Solving the allocation problem that maps an SLA on a virtualized execution platform is a high-level endeavor. Before this problem can be tackled appropriately, the issues related to performance impact of traffic processing module placement within hardware components must be studied and understood. Indeed, previous work [38] has shown that slight changes in processing pipeline configurations can result in vast swings in observed forwarding performance for network appliances.

We therefore first seek to characterize the performance of network traffic processing pipelines within stand-alone hardware processing unit. To this end, we have analyzed various packet I/O frameworks, polling strategies, pipeline execution models (task-to-core allocation strategies), allocation of networking hardware resources, and so on. This characterization gives insights on how to allocate networking functions onto a stand-alone machine, and leads to a low level component allocation framework called FastClick, which we describe in section 2.

We also study FastClick's fitness for purpose through the development of a novel network cloud function called SplitBox, a privacy preserving filtering function for outsourced firewalls (section 3). This application shows that FastClick automatic resource allocation algorithms on a stand-alone machine actually make a good use of available resources, providing good performance


for a demanding application, while requiring no advanced knowledge of the hardware platform from the network function implementor.

Next, we also investigate the characterization of a virtualized NFV infrastructure under different workloads in section 4.

Finally, section 5 hints at how our performance improvements on a bare-metal machine could translate into a virtualized setting.

2 Fast Userspace Packet Processing

In recent years, we have witnessed the emergence of high speed packet I/O frameworks, bringing unprecedented network performance to userspace. Using the Click modular router, we first review and quantitatively compare several such packet I/O frameworks, showing their superiority to kernel-based forwarding.

We then reconsider the issue of software packet processing, in the context of modern commodity hardware with hardware multi-queues, multi-core processors and non-uniform memory access. Through a combination of existing techniques and improvements of our own, we derive modern general principles for the design of software packet processors.

Our implementation of a fast packet processor framework, integrating a faster Click with both Netmap and DPDK, exhibits up-to about 2.3x speed-up compared to other software implementations, when used as an IP router.

2.1 Introduction

Recent years have seen a renewed interest for software network processing. However, as will be shown in section 2.3, a standard general-purpose kernel stack is too slow for line-rate processing of multiple 10-Gbps interfaces. To address this issue, several userspace I/O frameworks have been proposed. Those allow to bypass the kernel and obtain efficiently a batch of raw packets with a single syscall, while adding other capabilities of modern Network Interface Cards (NIC), such as support for multi-queuing.

This document first reviews the technical aspects of some existing userspace I/O frameworks, such as Netmap [19], Intel's DPDK [10], OpenOnload [22], PF_RING [16], PacketShader I/O [8] and Packet_MMAP [14]. Some other works go further than a "simple" Linux module bypassing the kernel, like IX [5] and Arrakis [17]. We won't consider them as, for our purpose, they only offer fast access to raw packets, but in a more protected way than the others I/O


frameworks, using virtualization techniques, and they were not fully available at the time this document was written.

To explore their performance inside a general purpose environment, we then compare the existing off-the-shelf integrations of some of these frameworks in the Click Modular Router [12]. Click allows to build routers by composing graphs of elements, each having a single simple function (e.g. decrementing a packet TTL). Packets then flow through the graph from input elements to output elements. Click offers a nice abstraction, includes a wealth of usual network processing elements, and already has been extended for use with some of the studied I/O frameworks. Moreover, we think its abstraction may lend itself well to network stack specialization (even if it's mostly router-oriented for now).

Multiple authors proposed enhancements to the Click Modular Router. RouteBricks [6] focuses on exploiting parallelism and was one of the first to use the multi-queue support of recent NICs for that purpose. However, it only supports the in-kernel version of Click. DoubleClick [11] focuses on batching to improve overall performances of Click with PacketShader I/O [8]. SNAP [23] also proposed a general framework to build GPU-accelerated software network applications around Click. Their approach is not limited to linear paths and is complementary to the others, all providing mostly batching and multi-queuing. All these works provide useful tips and improvements for enhancing Click, and more generally building an application on top of a "raw packets" interface.

The first part of our contribution is the critical analysis of those enhancements, and discuss how they interact with each other and with userspace I/O frameworks, both from a performance and a configuration ease points of view.

While all those I/O frameworks and Click enhancements were compared to some others in isolation, we are the first, to our knowledge, to conduct a global survey of their performance and, more importantly, interactions between the features they provides. Our contribution include new discoveries resulting from this in-depth factor analysis, such as the fact that the major part of performance improvement often attributed to batching is more due to the usage of a run-to-completion model, or the fact that locking can be faster than using multi-queue in some configurations.

Finally, based on this analysis, and new ideas of our own such as a graph analysis to discover the path that each thread can take to minimize the use of memory and multi-queue, we propose a new userspace I/O integration in Click (including a reworked implementation for Netmap, and a novel one for Intel's DPDK). Our approach offers both simpler configuration and faster performance. The network operator using Click does not need to handle low-level hardware-


related configuration anymore. Multi-queue, core affinity and batching are all handled automatically (but can still be tweaked). On the contrary to previous work which broke the compatibility by requiring a special handling of batches, our system is retro-compatible with existing Click elements. The Click developer is only required to add code where batching would improve performance, but it's never mandatory.

This new implementation is available at [4] and will conclude our contribution with the fastest version of Click we could achieve on our system, more flexible and easy-to-use with regard to modern techniques such as NUMA-assignement, multi-queuing, core and interrupt affinity or configuration of the underlying framework.

Section 2.2 reviews a set of existing userspace I/O frameworks. Section 2.3 then evaluates their forwarding performance. Section 2.4 discusses how some of these frameworks were integrated into the Click modular router. Section 2.5 analyzes those integrations, and various improvements to Click, giving insights for the design of fast userspace packet processors. We then propose FastClick, based on those insights. Finally, section 2.6 evaluates the performance of our implementation and section 2.7 concludes this section about FastClick.

2.2 I/O Frameworks

In this section, we review various features exhibited by most high performance userspace packet I/O frameworks. We then briefly review a representative sample of such frameworks.

2.2.1 Features

Zero-copy. The standard scheme for receiving and transmitting data to and from a NIC is to stage the data in kernel-space buffers, as one end of a Direct Memory Access (DMA) transfer. On the other end, the application issues read/write system calls, passing userspace buffers, which copy the data, across the protection domains, as a memory-to-memory copy.

Most of the frameworks we review aim to avoid this memory-to-memory copy by arranging for a buffer pool to reside in a shared region of memory visible to both NICs and userspace software. If that buffer pool is dynamic (i.e. the number of buffers an application can hold at any one time is not fixed), then true zero-copy can be achieved: an application which, for whatever reasons must hold a large number of buffers, can acquire more buffers. On the other hand, an application reaching its limit in terms of held buffers would then have


to resort to copying buffers in order not to stall its input loop (and induce packet drops).

Note however that some frameworks, designed for end-point applications, as opposed to a middlebox context, use separate buffer pools for input and output, thus requiring a memory-to-memory copy in forwarding scenarios.

Kernel Bypass. Modern operating system kernels provide a wide range of networking functionalities (routing, filtering, flow reconstruction, etc.). This generality does, however, comes at a performance cost which prevents to sustain line-rate speed in high-speed networking scenarios (either multiple 10-Gbps NICs, or rates over 10 Gbps). To boost performance, some frameworks bypass the kernel altogether, and deliver raw packet buffers straight into userspace. The main drawback of this approach is that this kernel bypass also bypasses the native networking stack; the main advantage is that the needed userspace network stack can be optimized for the specific scenario [13].

In pure packet processing applications, such as a router fast plane, a networking stack is not even needed. Note also that most frameworks provide an interface to inject packets “back” into the kernel, at an obvious performance cost, for processing by the native networking stack.

I/O Batching. Batching is used universally in all fast userspace packet frameworks. This is because batching amortizes, over several packets, the overhead associated with accessing the NIC (e.g. lock acquisition, system call cost, etc.).

Hardware multi-queue support. Modern NICs can receive packets in multiple hardware queues. This feature was mostly developed to improve virtualization support, but also proves very useful for load balancing and dispatching in multi-core systems. Indeed, for instance, Receiver-Side Scaling (RSS) hashes some pre-determined packet fields to select a queue, while queues can be associated with different cores.

Some NICs (such as the Intel 82599) also allow, to some extent, the explicit control of the queue assignment via the specification of flow-based filtering rules.

2.2.2 Frameworks

Packet_mmap [14] is a feature added to the standard UNIX sockets in the Linux kernel[1], using packet buffers in a memory region shared (mmaped, hence

1 When not mentioned explicitly, the *kernel* refers to Linux.


its name) between the kernel and the userspace. As such, the data does not need to be copied between protection domains. However, because Packet_mmap was designed as an extension to the socket facility, it uses separate buffer pools for reception and transmission, and thus zero-copy is not possible in a forwarding context. Also, packets are still processed by the whole kernel stack and need an in-kernel copy between the DMA buffer and the sk_buff, only the kernel to user-space is avoided and vice versa.

Framework	Packet mmap	Packet Shader I/O	Netmap	PF_RING ZC	DPDK	OpenOnload
Zero-copy	~	N	Y	Y	Y	Y
Kernel bypass	N	Y	Y	Y	Y	Y
I/O Batching	Y	Y	Y	Y	Y	Y
Hardware MQ	N	Y	Y	Y	Y	Y
Device families supported	All	1	ZC: 8 No ZC: all	ZC: 4 No ZC: all	11	SolarFlare
PCAP Library	Y	N	Y	Y	Y	Y
License	GPLv2	GPLv2	BSD	Proprietary	BSD	Proprietary
IXGBE version	Last	2.6.28	Last	Last	Last	N/A

Table 2: I/O frameworks features summary.

PacketShader [8] is a software router using the GPU as an accelerator. For the need of their work, the authors implemented PacketShader I/O, a modification of the Intel IXGBE driver and some libraries to yield higher throughput. It uses pre-allocated buffers, and supports batching of RX/TX packets. While the kernel is bypassed, packets are nevertheless copied into a contiguous memory region in userspace, for easier and faster GPU operations.

Netmap [19] provides zero-copy, kernel bypass, batched I/O and support for multi-queuing. However, the buffer pool allocated to an application is not dynamic, which could prevent true zero-copy in some scenarios where the application must buffer a lot of packets. Recently, support for pipes between applications has also been added. Netmap supports multiple device families (IGB, IXGBE, r8169, forcedeth, e1000 and e1000e) but can emulate its API over any driver at the price of reduced performance.

PF_RING ZC (ZeroCopy) [16] is the combination of ntop's PF_RING and ntop's DNA/LibZero. It is much like Netmap [15], with both frameworks evolving in the


same way, adding support for virtualization and inter-process communication. PF_RING ZC has also backward compatibility for non-modified drivers, but provides modified drivers for a few devices. The user can choose to detach an interface from the normal kernel stack or not. As opposed to Netmap, PF_RING ZC supports huge pages and per-NUMA node buffer regions, allowing to use buffers allocated in the same NUMA node than the NIC.

A major difference is that PF_RING ZC is not free while Netmap is under a BSD-style license. The library allows 5 minutes of free use for testing purpose, allowing us to do the throughput comparison of section 2.4 but no further testing. Anyway, the results of this paper should be applicable to PF_RING DNA/ZC.

DPDK. The Intel Data Plane Development Kit [10] is somehow comparable to Netmap and PF_RING ZC, but provides more user-level functionalities such as a multi-core framework with enhanced NUMA-awareness, and libraries for packet manipulation across cores. It also provides two execution model: a pipeline model where typically one core takes the packets from a device and give them to another core for processing, and a run-to-completion model where packets are distributed among cores using RSS, and processed on the core which also transmits them.

DPDK can be considered more than just an I/O framework as it includes a packet scheduling and execution model. However, DPDK exhibits fewer features than the Click modular router.

DPDK originally targeted, and is thus optimized for, the Intel platform (NICs, chipset, and CPUs), although its field of applicability is now widening.

OpenOnload [22] is comparable to DPDK but made by SolarFlare, only for their products. It also includes a user-level network stack, allowing to quickly accelerate existing applications.

We do not consider OpenOnload further in this paper, because we do not have access to a SolarFlare NIC.

Table 2 summarize the features of the I/O frameworks we consider.

2.3 Pure I/O forwarding evaluation

For testing the I/O system we used a computer running Debian GNU\Linux using a 3.16 kernel on an Intel Core i7-4930K CPU with 6 physical cores at 3.40 GHz, with hyper-threading enabled [9]. The motherboard is an Asus P9X79-E WS [2]with 44 GB of RAM at 1.6 GHz in Quad-Channel mode. We use 2

Intel (dual port) X520 DA cards for our performances tests. Previous experiments showed that those Intel 82599-based cards cannot receive small packets at line-rate, even with the tools from framework's author [8][19]. Experiments done for figure 1 lead to the same conclusion, our system seems to be capped to 33 Gbps with 64-byte packets.

To be sure that differences in performances is due to changes in the tested platform, a Tilera TileENCORE Gx36 card fully able to reach line-rate in both receive and transmit side was used. We used a little software of our own available at [4] to generate packets on the Tilera at line-rate towards the computer running the tested framework connected with 4 SFP+ DirectAttach cables, and count the number of packets received back. All throughput measurements later in this paper indicate the amount of data received back in the Tilera, 40 Gbps meaning that no loss occurred, and a value approaching 0 that almost all packets were lost. We start counting after 3 seconds to let the server reach stable state and compute the speed for 10 seconds. The packets generated have different source and destination IP addresses, to enable the use of RSS when applicable.

For these tests there is no processing on the packets: packets turning up on a specific input device are all forwarded onto a pre-defined, hardwired output device. Each framework has been tuned for its best working configuration including batch size, IRQ affinity, number of cores, thread-pinning and multi-queue. All forwarding tests were run for packet sizes varying from 60 to 1024 bytes, excluding the 4 bytes of the Frame Check Sequence appended at the end of each Ethernet frame by the NIC.

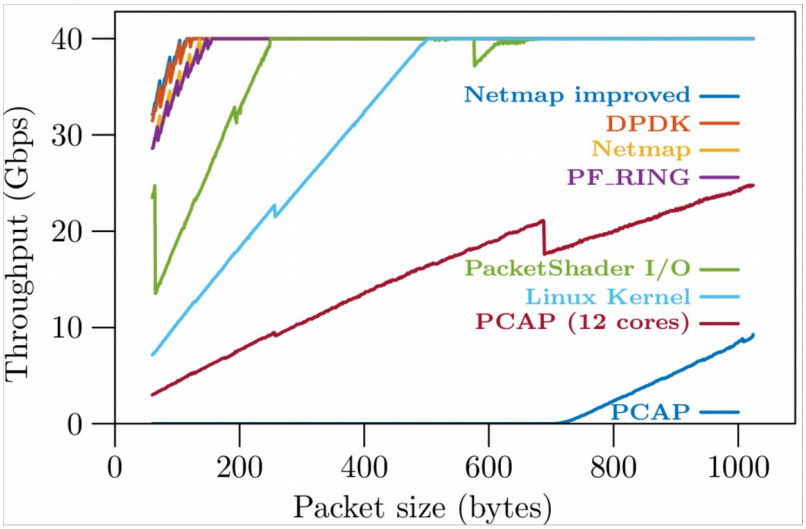


Figure 1: Forwarding throughput for some I/O frameworks using 4 cores and no multi-queuing.


We realized that our NICs have a feature whereby the status of a transmit packet ring can be mirrored in memory at very low performance cost. We modified Netmap to exploit this feature and also limited the release rate of packets consumed by the NIC to only recover buffer for sent packets once every interrupt, which released the PCIe bus of useless informations and brought Netmap performances above DPDK as we can see in figure 1. For 64-byte packets, theses improvements boost the throughput by 14%, 1.5% over DPDK. However, except for the line labeled “Netmap Improved” in figure 1, only the final evaluation in section 2.6 use this improved Netmap version.

As expected (because they share many similarities), PF_RING has performance very similar to (standard) Netmap as shown in figure 1.

The wiggles seen in DPDK, Netmap and PF_RING are due to having two NICs on the same Intel card and produce a spike every 16 bytes. When using one NIC per card with 4 PCIe cards the wiggles disappears and performance is a little better. The wiggles have a constant offset of 4 bytes with PF_RING, but we couldn’t explain it, mainly because PF_RING sources are unavailable.

PacketShader I/O is slower because of its userspace copy, while the other frameworks that use zero-copy do not even touch the packet content in theses tests and do not pay the price of a memory fetching.

We also made a little Linux module available at [4] which transmits all received packets on an interface back to the same interface directly within the interrupt context. It should show the better performances that the kernel is able to provide in the same 4 cores and no multi-queue conditions. The relative slowness of the module regarding the other frameworks can be explained by the fact the receive path involves the building of the heavy skbuff and the whole path involves multiple locking, even if in our case no device is shared between multiple cores.

PCAP shows very poor performances because it does not bypass the kernel like the other frameworks and, in addition to the processing explained for the Linux module, the packet needs to go through through the kernel forward information base (FIB) to find its path to the PCAP application. But the bigger problem is that it relies on the kernel to get packets, and each IRQ cause too much processing by the kernel, which is overwhelming a single core and does not let enough time for any thread actually consuming the packets to run, even with NAPI[2] enabled.

2 NAPI, which stands for “New API” is an interface to network device drivers designed for high-speed networking via interrupt mitigation and packet throttling [21].


This is known as a receive live-lock and [20] shows that beyond a certain packet rate the performance drops. The solution is either to use a polling system like in FreeBSD [18] or DPDK, or reduce the “per-packet” cost in the IRQ routines like in Netmap where it does very little processing (e.g. flags and ring buffer counter updates), or to distribute the load using techniques like multi-queue and RSS to ensure that each core receives fewer packets than the critical live lock threshold. Our kernel module is not subject to the receive live lock problem because the forwarding of the packet is handled in the IRQ routine which does not interrupt itself.

To circumvent the live lock problem seen with PCAP, we used the 12 hyper-threads available on our platform and 2 hardware queue per device to dispatch interrupt requests on the first 8 logical cores, while 4 PCAP threads forwards packets on their own last 4 logical cores. The line labeled “PCAP (12 cores)” shows the results of that configuration which gave the best results we could achieve out of many possible configurations exchanging the number of cores to serve the IRQ and the PCAP threads.

However this setup is using all cores at 100% and still provides performances way below the previous frameworks which achieve greater throughput even when using only one core.

2.4 I/O integrations in Click

We chose the Click modular router to build a fast userspace packet processor. We first compare several systems integrating various I/O frameworks with either a vanilla Click, or an already modified Click for improved performance.

In Click, packet processors are built as a set of interconnected processing elements. More precisely, a Click task is a schedulable chain of elements that can be assigned to a Click thread (which, in turn, can be pinned to a CPU core). A task always runs to completion, which means that only a single packet can be in a given task at any time. A Click forwarding path is the set of elements that a packet traverses from input to output interfaces, and consists of a pipeline of tasks interconnected by queues.

While Click itself can support I/O batching if the I/O framework exposes batching (a FromDevice element can pull a batch of packets from an input queue), the vanilla Click task model forces packet to be processed individually by each task, with parallelism resulting from the chaining of several tasks.

Also, vanilla Click uses its own packet pool, copying each packet to and from the buffers used to communicate with the interface (or hardware queue). As


such, even if the I/O framework supports zero-copy, vanilla Click imposes two memory-to-memory copies.

For our tests, we use the following combinations of I/O framework-Click integration:

- Vanilla Click + Linux Socket: this is our off the shelf baseline configuration. The Linux socket does not expose batching, so I/O batching is not available to Click.
- Vanilla Click + PCAP: while PCAP eliminates the kernel to userspace copy by using packet_mmap, Click still copies the packets from the PCAP userspace buffer into its own buffers. However, PCAP uses I/O batching internally, only some of the library calls from Click produce a syscall.
- Vanilla Click + Netmap: as netmap exposes hardware multi-queues, these can appear as distinct NICs to Click. Therefore multi-queue configuration can be achieved by using one Click element per hardware queue. Netmap exposes I/O batching, so Click uses it.
- DoubleClick [11]: integrates PacketShader I/O into a modified Click. The main modification of Click is the introduction of compute batching, where batches of packets (instead of individual packets) are processed by an element of a task, before passing the whole batch to the next element. PacketShader I/O exposes I/O batching and supports multi-queuing.
- Kernel Click: To demonstrate the case for userspace network processing, we also run the kernel-mode Click. We only modified it to support the reception of interrupts to multiple cores. Interrupt processing (creating a skbuff for each incoming packets) is very costly and using multiple hardware queues pinned to different cores allows to spread the work. Our modification has been merged in the mainline Click [3].

  Kernel Click had a patch for polling mode, but it’s only for e1000 driver and only supports very old kernels which prevent our system from running correctly.

	NETMAP	PCAP	UNIX SOCKETS	DOUBLE CLICK	KERNEL	FASTCLICK NETMAP	FASTCLICK DPDK
I/O framework	Netmap	PCAP	Linux sockets	PSIO	Linux kernel	Netmap	DPDK
IO batching	Y	N	N	Y	N	Y	Y
Computation batching	N	N	N	Y	N	Y	Y
MQ support	Y	N	N	Y	N	Y	Y
No copy inside Click	N	N	N	Y	N	Y	Y

Table 3: Click integrations of I/O frameworks.

Table 2 summarizes the features of these I/O framework integrations into Click.

We ran tests for pure packet forwarding, similar to those in section 2.2, but through Click. Each packet is taken from the input and transmitted to the output of the same device. The configuration is always a simple FromDevice pushing packets to a ToDevice. These two elements must be connected by a queue, except in DoubleClick where the queue is omitted (and thus the FromDevice and ToDevice elements run in the same task) because PacketShader I/O does not support the select operation. As a result, the ToDevice in DoubleClick cannot easily check availability of space in the output ring buffer, while the FromDevice continuously polls the input ring buffer for packets. As soon as the FromDevice gets packets, these are thus completely processed in a single task.

While this scenario is somewhat artificial, it does provide baseline ideal (maximum) performance.

In all scenarios, corresponding FromDevice and ToDevice are pinned to the same core. The results are shown in figure 2. For this simple forwarding case, compute batching does not help much as the Click path consists of a very small pipeline and the Netmap integration already takes advantage of I/O Batching. Therefore Netmap closely follows DoubleClick.

The in-kernel Click, the integration of Click with PCAP and the one using Linux Socket all showed the same receive live lock behavior than explained in section 2.3. The same configurations where interrupt requests (IRQ) are dispatched to 8 logical cores and keep 4 logical cores to run Click lead to the best performances for those 3 frameworks.

The in-kernel Click is running using kernel threads and is therefore likely to receive live lock for the same reason than the PCAP configuration in section 2.3. The interrupts do less processing than for sockets because they do

not pass through the forward information base (FIB) of the kernel but still create heavy skbuff for each packet and call the “packet handler” function of Click with a higher priority than Click’s thread themselves, causing all packet to be dropped in front of Click’s queue while nearly never servicing the queue consumer.

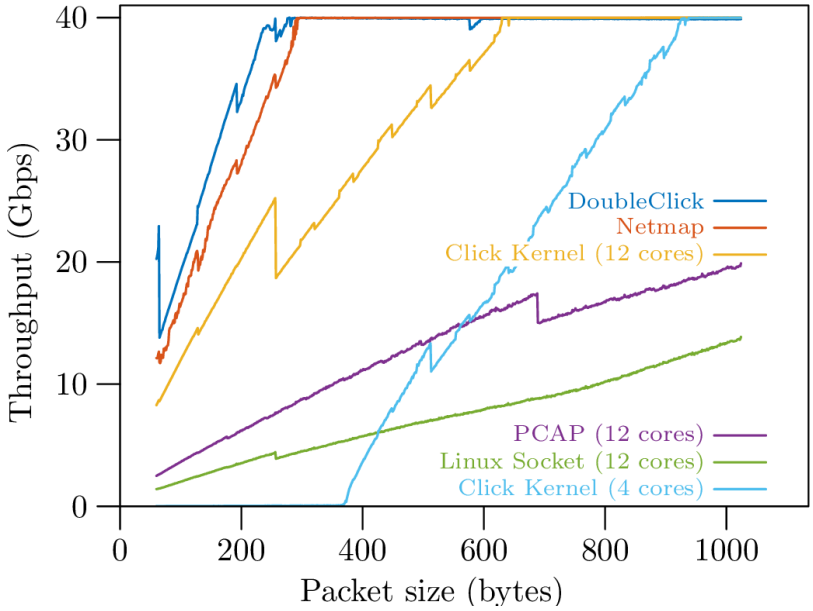


Figure 2: Forwarding throughput for some Click I/O implementations with multiples I/O systems using 4 cores except for integration heavily subject to receive live lock.

We also tested a router configuration similar to the standard router from the original Click paper, changing the ARP encapsulation element into a static encapsulation one. Each interface represents a different subnetwork, and traffic is generated so that each interface receives packets destined equally to the 3 others interfaces, to ensure we can reach line-rate on output links. As the routing may take advantage of flows of packets, having routing destination identical for multiple packets, our generator produces flows, that is packets bearing an identical destination, of 1 to 128 packets. The probability of a flow size is such that small flows are much more likely than large flows (fig. 3).

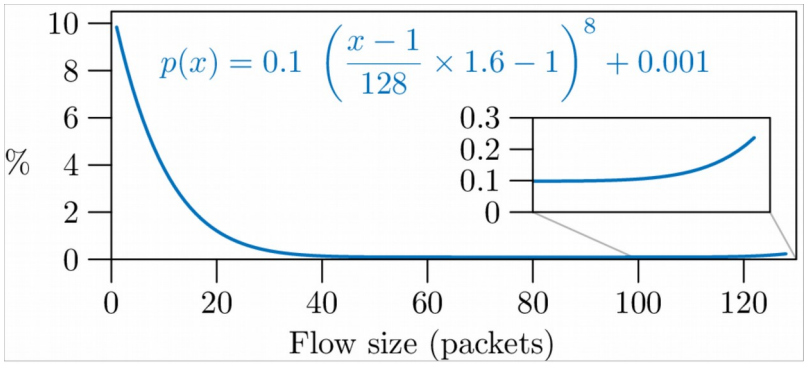



Figure 3: Probability of having flow of 1 to 128 packets for the router packet generator.

Results are shown in figure 4. We omit the PCAP and socket modes as their performance is very low in the forwarding test. Additionally, we show the Linux kernel routing functionality as a reference point.

DoubleClick is faster than the Netmap integration in Click, owing to its compute batching mode and its single task model. They are both faster than the Linux native router as it does much more processing to build the skbuffs and go through the FIB than Click which does only the minimal amount of work for routing. The Kernel-Click is still subject to receive live lock and is slower than the native kernel router when routing is done on only 4 cores. Even when using 12 cores, Kernel-Click is slower than DoubleClick and the Netmap integration.

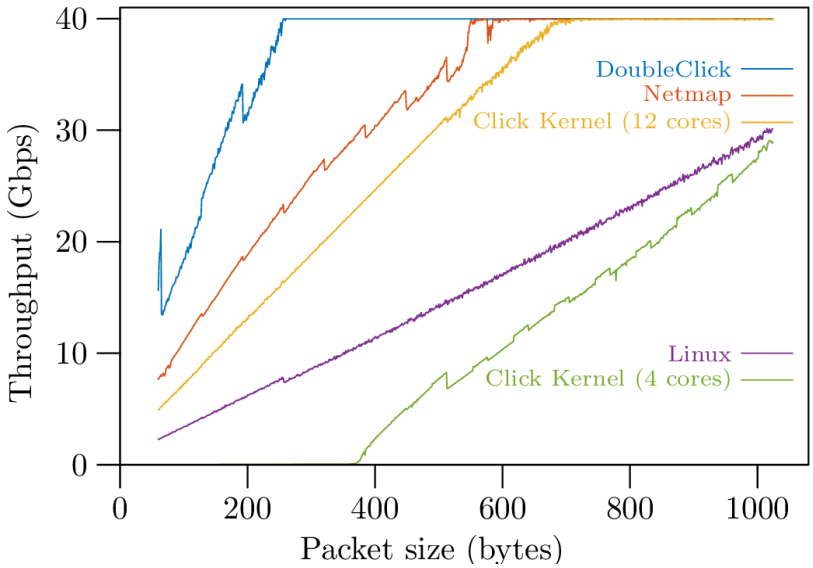


Figure 4: Throughput in router configuration using 4 cores except for in-kernel Click.

2.5 Analysis towards FastClick

We now present an in-depth analysis of the Click integrations, discussing their pros and cons. This will ultimately lead to general recommendations for the design and implementation of fast userspace packet processors. As we implement these recommendations into Click, we refer to them as FastClick for convenience.

In fact, we integrated FastClick with both DPDK and Netmap. DPDK seems to be the fastest in term of I/O throughput, while Netmap affords more fine-grained control of the RX and TX rings, and already has multiple implementations on which we can build upon and improve.

The following section starts from vanilla Click as is. Features will be reviewed and added one by one. Starting from here, FastClick is the same than vanilla Click, without compute batching, or proper support for multi-queuing.

2.5.1 I/O Batching

Both DPDK and Netmap can receive and send batches of packets, without having to do a system call after each packet read or written to the device. Figure 5 assess the impact of I/O batching using Click in a modified version to force the synchronization call after multiple batch size in both input and output.

Vanilla Click always process all available packets before calling again Netmap's poll method – the poll method indicates how many packets are in the input queue. It reads available packets in batches and transmits it as a burst which is a series of transmission one packet at a time through a sequence of Click elements. The corresponding tasks only relinquishes the CPU at the end of the burst. Vanilla Click will reschedule the task if any packet could be received. On the other hand FastClick will only reschedule the task if a full I/O burst is available. This strategy tends to forces the use of bigger batches and thus preserves the advantages of I/O batching.

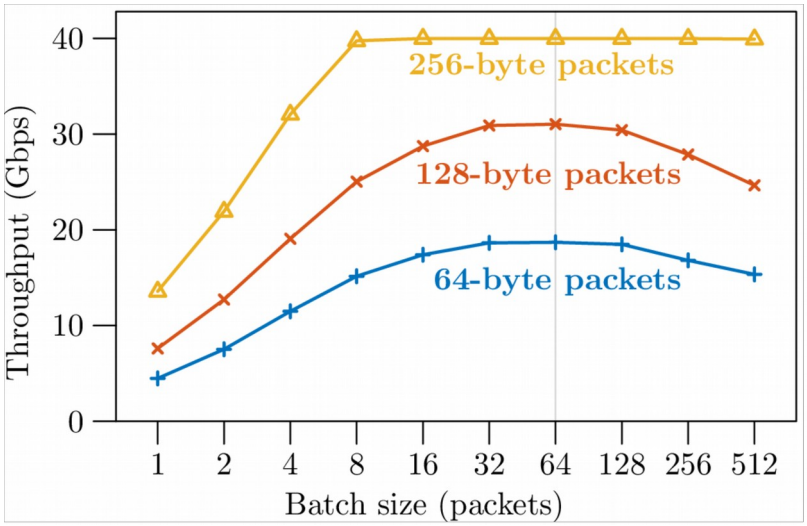


Figure 5: I/O Batching – Vanilla Click using Netmap with 4 cores using the forwarding test case (queue size = 512). A synchronization is forced after each "batch size".

2.5.2 Ring Size

The burst limit is there for insuring that synchronization is not done after too few packets. As such it should not be related to the ring size. To convince ourselves, we ran the same test using FastClick with multiple ring sizes and

found that the better burst choice is more or less independent to the ring size as shown in figure 6.

What was surprising though is the influence of the ring size on the performances.

With bigger ring size, the amount of CPU time spent in memcpy function to copy the packet's content goes from 4% to 20%, indicating that the working set is too big for the CPU's cache.

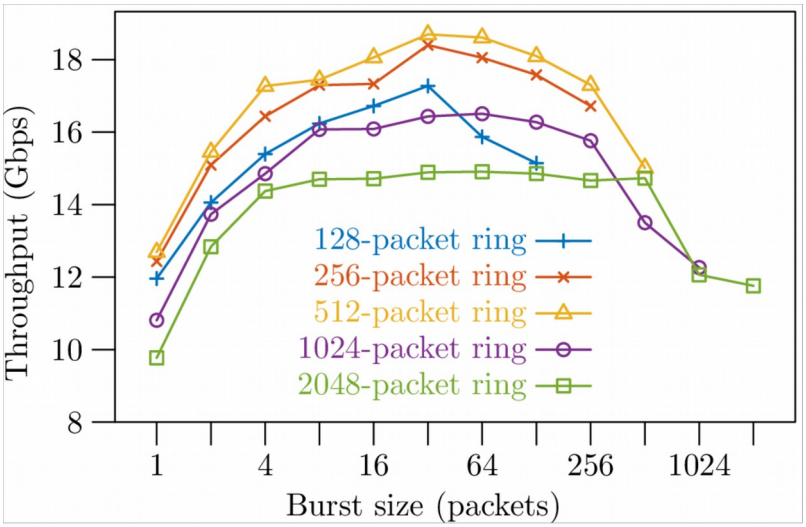


Figure 6: Influence of ring size - FastClick using Netmap with 4 cores using the forwarding test case and packets of 64 bytes.

2.5.3 Execution Model

In the standard Click, all packets are taken from an input device and stored in a FIFO queue. This is called a "push" path as packets are created in the "FromDevice" elements and pushed until they reach a queue. When an output "ToDevice" element is ready (and has space for packets in the output packet ring it feeds), it traverses the pipeline backwards asking each upstream element for packets. This is called a "pull" path as the ToDevice element pulls packets from upstream elements. Packets are taken from the queue, traverse the elements between the queue and the ToDevice and are then sent for transmission as shown in figure 7 (a).

One advantage of the queue is that it divides the work between multiple threads, as one thread can take care of the part between the FromDevice and the queue, and another thread can handle the path from the queue to the ToDevice. Another advantage is that the queue allows the ToDevice to receive packets only when it really has space to receive packets. It will only call the pull

path when it has some space and, when I/O batching is supported, for the amount of available space.

But there are two drawbacks. First, if multiple threads can write to the queue, some form of synchronization must be used between these threads, resulting in some overhead. Second, if the pushing thread and the pulling thread run on different cores, misses can occur at various levels of the cache hierarchy, resulting in a performance hit as the packets are transfered between cores.

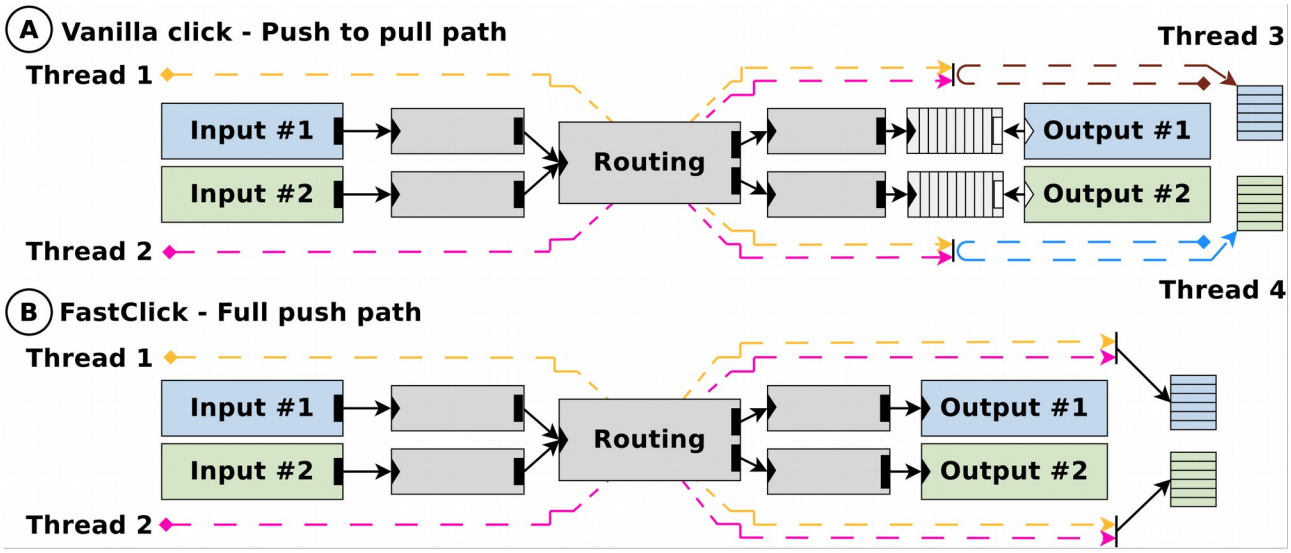


Figure 7: Push to Pull and Full Push path in Click.

Therefore, we adopt a model without queue: the full-push model where packets traverse the whole forwarding path, from FromDevice to ToDevice, without interruption, driven by a single thread.

NICs now possess receive and transmit rings with enough space to accommodate up to 4096 packets for the Intel 82599-based cards (not without some cost as seen in section 2.5.2), so these are sufficient to absorb transient traffic and processing variations, substituting advantageously for the Click queues.

Packets are taken from the NIC in the FromDevice Element and packets are pushed one by one into the Click pipe, even if I/O batching is supported. It does so until it reaches a ToDevice Element and adds it in the transmit buffer as shown in figure 7 (b). If there is no empty space in the transmit ring, the thread will either block or discard the packets.

This method is introducing 3 problems.

- All threads can end up in the same output elements. So locking must be used in the output element before adding a packet to the output ring.

- Depending on packet availability at the receive side, packets are added to the output rings. But the output ring must be synchronized sometime, to tell the NIC that it can send some packets. Syncing too often cancels the gain of batching, but syncing too sporadically introduces latency.

  DPDK forces a sync every maximum 32 packets, while Netmap does it for every chosen I/O burst size.

- When the transmit ring is full, two strategies are possible: the output has a blocking mode, doing the synchronization explained above until there is space in the output ring; in non blocking mode, the remaining packets are stored in a per-thread internal queue inside the ToDevice, dropping packets when the internal queue is longer than some threshold.

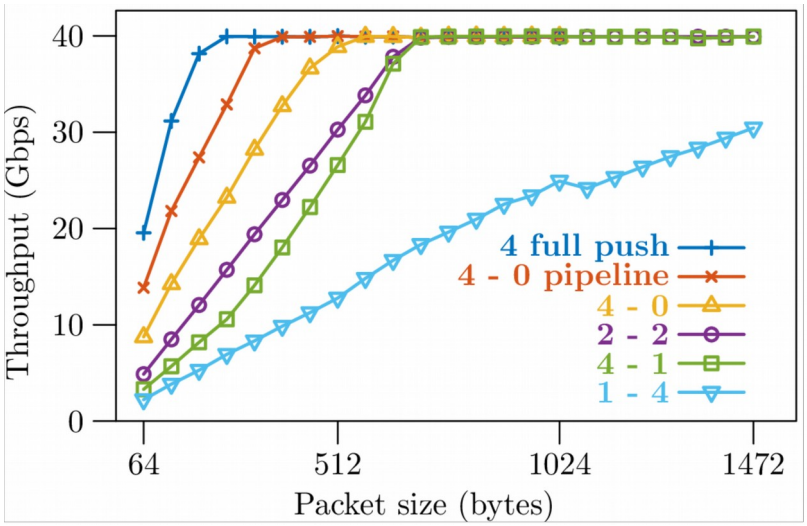


Figure 8: Comparison between some execution models. Netmap implementation with 4 or 5 cores using the router test case.

Figure 8 shows a performance comparison using the Netmap implementation with I/O batching, and a varying number of cores running FromDevice and ToDevice (the label $i-j$ represents i cores running the 4 FromDevice and j cores running the 4 ToDevice). The full push, where we have a FromDevice and all the ToDevice in a single thread on each core, performs best. The second best configuration corresponds to also a FromDevice and all the ToDevice running on the same core, but this time as independent Click tasks with a Click queue in between (label 4-0). Even when using 5 cores, having one core taking care of the input or the output expectedly results in a CPU constrained configuration.

While full-push mode seems best for our usage, one could need the “pipeline” mode anyway, having one thread doing only one part of the job, by using Queue element. But even in this case the full push execution model prove to be


faster as shown in figure 8 by the line labeled “4 - 0 pipeline”. Using our new Pipeliner element which can be inserted between any two Click elements there is no more “pull” path in the Click sense. Packets are enqueued into a per-thread queue inside the pipeliner element and it is the thread assigned to empty all the internal queues of the pipeliner element which drives the packet through the rest of the pipeline.

Full-push path is already possible in DoubleClick but only as a PacketShader I/O limitation and we wanted to study further its impact and why it proves to be so much faster by comparing the Netmap implementation with and without full push, decoupling it from the introduction of compute batching.

In vanilla Click, a considerable amount of time is spent in the notification process between the Queue element and the ToDevice. It reaches up to 60% of CPU time with 64-byte packets for the forwarding test case. With full push path, when a packet is received, it is processed through the pipeline and queued in the Netmap output ring. When the amount of packets added reaches the I/O batch size or when the ring is full, the synchronization operation explained above is called. The synchronization takes the form of an ioctl with Netmap, which updates the status of the transmit ring so that available space can be computed. This also allows a second improvement as the slower select/poll mechanism isn’t used anymore for the transmit side, not having to constantly remove and add the Netmap file descriptor to Click’s polling list anymore.

Click allows to clone packets by keeping a reference to another packet as the one containing the real data, using a reference counter to know when a data packet can be freed. In the vanilla Click, the packet can be cloned and freed by multiple cores, therefore an atomic operation has to be used to increment and decrement the reference counter. In full push mode, we know that it is always the same core which will handle the packets, therefore we can use normal increment and decrement operations instead of the atomic ones. That modification showed a 3% improvement with the forwarding test case and a 1% improvement with the router test case. FastClick automatically detect a full push configuration using the technique described in the section 2.5.6.

Because DPDK does not support interrupts (and it must therefore poll continuously its input), our DPDK integration only supports full-push mode.

2.5.4 Zero Copy

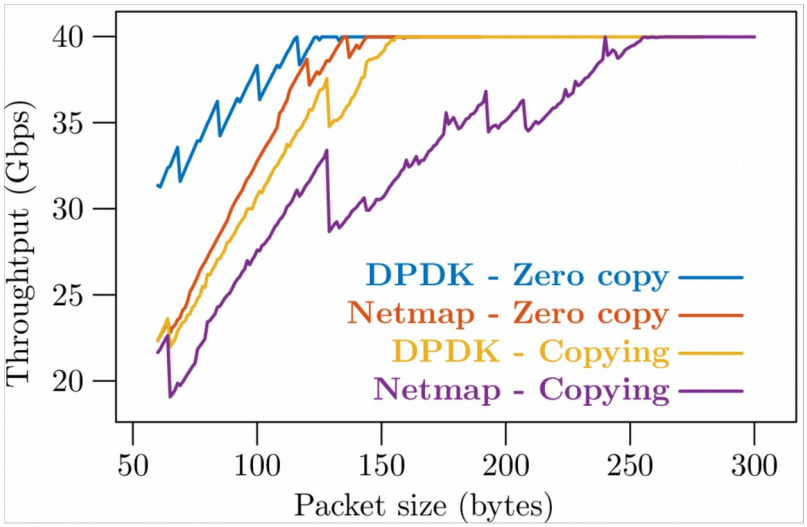


Figure 9: Zero copy influence - DPDK and Netmap implementations with 2 cores using the forwarding test case.

Packets can be seen as two parts: one is the packet meta-data which in Click is the Packet object and is 164-byte as of today. The other part is the packet data itself, called the buffer which is 2048 bytes both for Click and Netmap. The packet meta-data contains the length of the actual data in the buffer, the time-stamp and some annotations to the packet used in the diverse processing functions.

In the vanilla Click, a buffer space is needed to write the packet's content, but allocating a buffer for each freshly received packets with malloc() would be very slow. For this reason, packets are pre-allocated in "pools". There is one pool per thread to avoid contention problems. Pools contain pre-allocated packet objects, and pre-allocated buffers. When a packet is received, the data is copied in the first buffer from the pool, and the meta-data is written in the first packet object. The pointer to the buffer is set in the packet object and then it can be sent to the first element for processing.

Netmap has the ability to swap buffer from the receive and transmit rings. Packets are received by the NIC and written to a buffer in the receive ring. We can then swap that buffer with another one to keep the ring empty and do what we want with the filled buffer. This is useful as some tasks such as flow reconstruction may need to buffer packets while waiting for further packet to arrive. By allocating a number of free buffers and swapping a freshly received packet with a free buffer, we can delay processing of the packet while not keeping occupied slots in the receive ring. This also allows to swap buffers with the transmit ring, allowing "zero-copy" forwarding, as a buffer is never copied.

We implemented an ioctl using the technique introduced in SNAP [23] to allocate a number of free buffers from the Netmap buffer space. A pool of Netmap buffers is initialized, substituted for the Click buffer pool. When a packet is received, the Netmap buffer from the receive ring is swapped for one of the buffers from the Click buffer pool.

DPDK directly provide a swapped buffer, as such we can directly give it to the transmit ring instead of copying its content.

With Netmap, the update of the transmit ring is so fast that output ring is nearly always full, and the ioctl to sync the output ring is called too often, especially its part to reclaim the buffers from packets sent by the NIC which is very slow and is forced to be done when using the ioctl. Instead, we changed two lines in Netmap to call the reclaim part of the ioctl only if a transmit side interrupt has triggered. We set Netmap's flag "NS_REPORT" one packet every 32 packets to ask for an interrupt when that packet has been sent.

We used the forwarding test case as our first experiment, with only two cores to serve the 4 NICs to ensure that the results are CPU-bound, and that better performance is indeed due to a better packet system. The results are shown in figure 9. The test case clearly benefits from zero-copy, while copy mode produces more important drops in the graph as one byte after the cache line size forces further memory accesses.

2.5.5 Multi-queuing

Multi-queuing can be exploited to avoid locking in the full-push paths. Instead of providing one ring for receive and one ring for transmit, the newer NICs provide multiple rings per side, which can be used by different threads without any locking as they are independent as shown in figure 10.

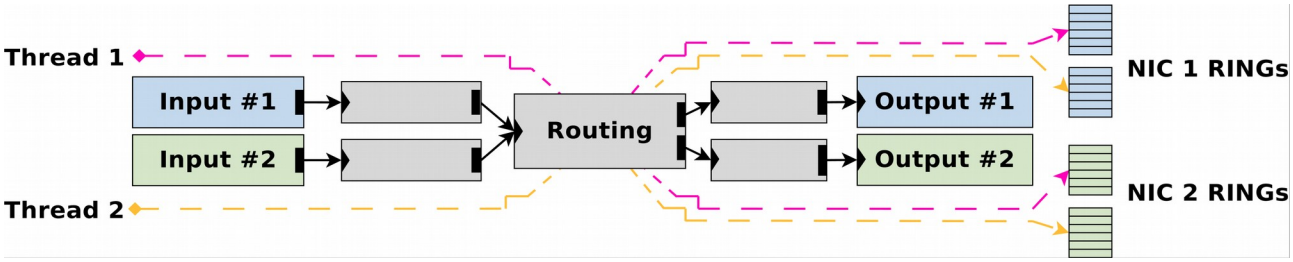


Figure 10: Full push path using multi-queue.

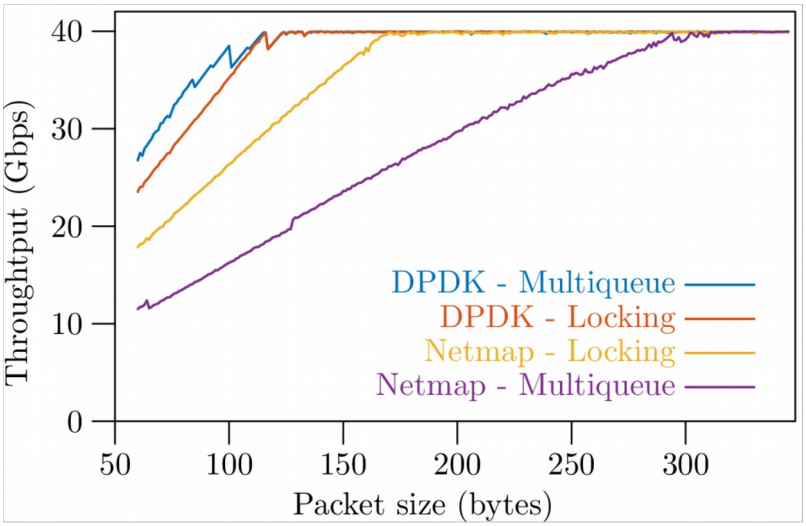


Figure 11: Full push path using multi-queue compared to locking - DPDK and Netmap implementations with 4 cores using the router test case.

We compared the full push mode using locking and using multiple hardware queues both with DPDK and Netmap, still with 4 cores. Netmap cannot have a different number of queues in RX and TX, enabling 4 TX queues forces us to look for incoming packets across the 4 RX queues. The results are shown in figure 11.

The evaluation shows that using multiple queues is slower than locking using Netmap, but provides a little improvement with DPDK. With Netmap, augmenting the number of queues produces the same results than augmenting the number of descriptors per rings, as seen in section 2.5.2. Both ends up multiplying the total number of descriptors, augmenting the size of Click's working set to the point where it starts to be bigger than our CPU's cache. As we use zero-copy, we see that the cost of reading and writing from and to Netmap's descriptors goes up with the number of queues.

2.5.6 Handling Mutable Data

In the vanilla Click one FromDevice element takes packets from one device. As show in section 2.3, handling 10 Gbps of minimal-size packets on one core is only possible with a fast framework to deliver quickly packets to userspace and a fast processor. And even in this configuration, any processing must be delegated to another core as the core running the FromDevice is nearly submerged by the reception. A solution is to use multi-queuing, not only to avoid locking like for the full push mode, but to exploit functionality such as Receive Side Scaling (RSS) which partitions the received packets among multiple queues, and therefore enables the use of multiple cores to receive packets from the same device. However, packets received in different hardware


queues may follow the same Click path. The question is thus how to duplicate the paths for each core and how to handle mutable state, that is, per-element data which can change according to the packets flowing through it, like caches, statistics, counters, etc. In figure 12, the little dots in Routing elements represent per-thread duplicable meta-data, like the cache of a last seen IP route, and the black big dot is the data which should not be duplicated because either it is too big, or it needs to be shared between all packets (like an ARP Table, a TCP flow table needing both direction of the flow, etc).

In a multi-queue configuration, there will be one FromDevice element attached to one input queue. Each queue of each input device is handled by different cores (if possible), otherwise some queues are assigned to the same thread. The problem is that in most cases, the Click paths cross at one element that could have mutable data.

A first solution is to use thread-safe elements on the part of the path that multiple threads can follow as in figure 12 (a). Only mutable data is duplicated, such as the cache of recently seen routes per cores but not the other non-mutable fields of the elements such as the Forward Information Base (FIB). The advantage of this method is that in some cases memory duplication is too costly, for example in the case of a big FIB, although the corresponding data structure must become thread safe. Moreover, the operator must use a special element to either separate the path per-thread to use one output queue for each thread, or use a thread-safe queue and no multi-queue.

This is in contrast to the way SNAP and DoubleClick approach the issue: the whole path is completely duplicated, as in figure 12 (b).

A third solution would be to duplicate the element for each thread path with a shared pointer to a common un-mutable data like in figure 12 (c). But that would complicate the Click configuration as we would need to instantiate one element (let’s say an IP router) per thread-path, each one having their own cache, but pointing to a common element (the IP routing table).

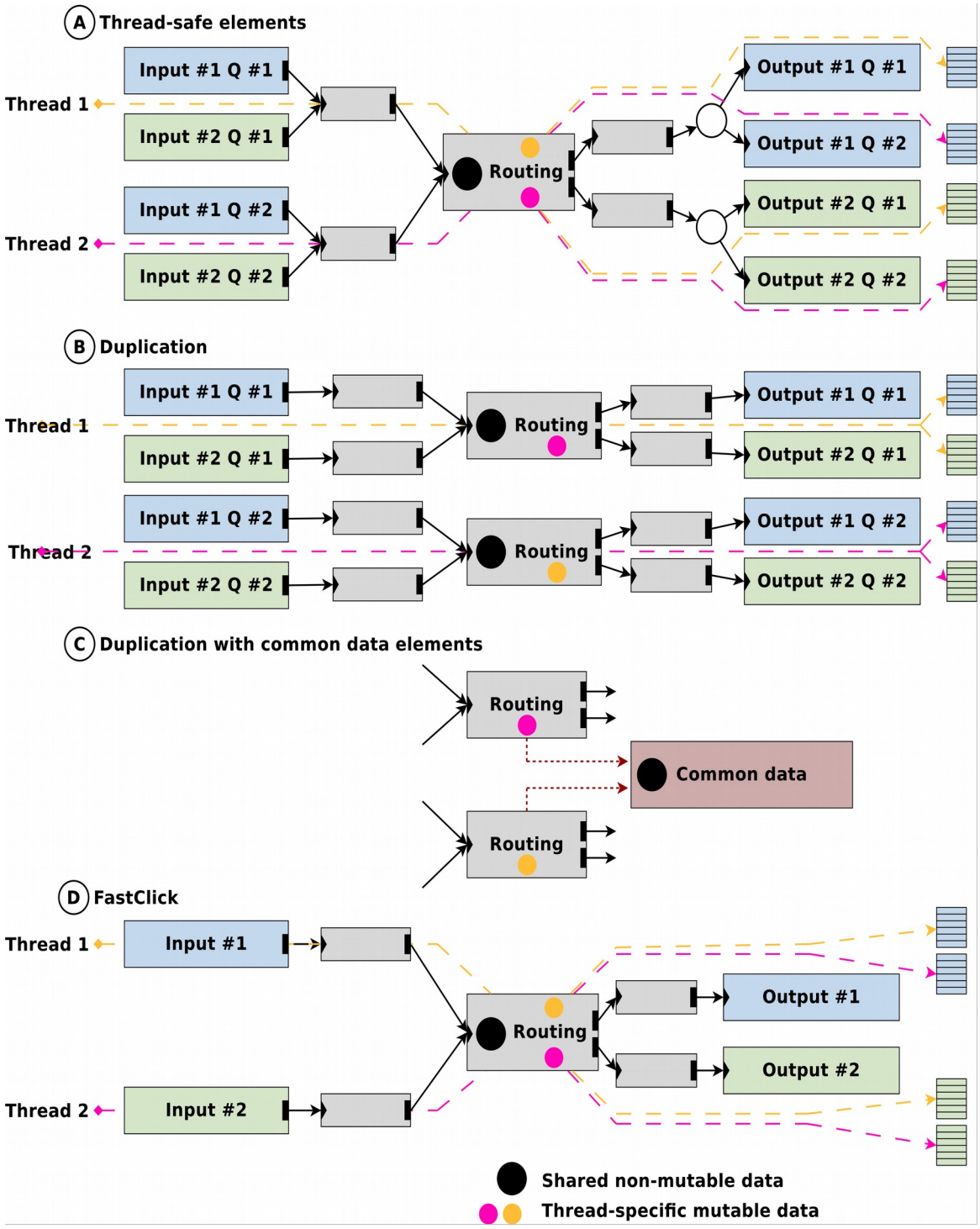


Figure 12: Three ways to handle data contention problem with multi-queue and our solution.

We prefer to go back to the vanilla Click model of having one Element per input device and one per output device. In that way the user only cares about the logical path. Our FastClick implementation takes care of allocating queues and threads in a NUMA-aware way by creating a Click task for each core allocated for the input device, without multiplying elements. It transparently manage the


multi-queuing by assigning one hardware queue per thread as in figure 12 (d) so the operator do not need to separate path according to threads or join all threads using a queue as in figure 12 (a).

Of course, the user can still specify parameter to the FromDevice to change the number of assigned threads per input device. He can also force each thread to take care of one queue of each device to allow the same scheme than in figure 12 (a) but still with one input element and one output element per device. The difference between the two configurations depends mostly on the use case. Having each core handling one queue of each device allows more load-balancing if some NICs have less traffic than others, but if the execution paths depend strongly on the type of traffic, it could be better to have one core doing always the same kind of traffic and avoid instruction cache misses.

To assign the threads to the input device we do as follows: We use only one thread per core. For each device, we identify its NUMA node and count the number of devices per NUMA node. We then divide the number of available CPU cores per NUMA node by the number of devices on that NUMA node, which gives the number of cores (and thus threads) we can assign to each device. With DPDK, we use one queue per device per thread, but with Netmap we cannot change the number of receive queues (which must be equal to the number of send queues), and have to look across multiple queues with the same thread if there are too many queues.

For the output devices, we have to know which threads will eventually end up in the output element corresponding to one device, and assign the available hardware queues of that device to those threads. To do so, we added the function getThreads() to Click elements. That function will return a vector of bits, where each bit is equal to 1 if the corresponding thread can traverse this element, that is called the thread vector.

FastClick performs a router traversal at initialization time, so hardware output queues are assigned to threads.

To do so, the input elements have a special implementation of getThreads() to return a vector with the bits corresponding to their assigned threads set to 1. For most of the other elements, the vector returned is the logical OR of the vector of all their input elements, because if two threads can push packets to a same element, this element will be executed by either of these threads. Hence, this is the default behavior for an elements without specific getThreads() function.

An example is shown in figure 13. In that example, some path contains a queue where multiple threads will push packets. As only one thread takes packet from the top right queue, the output #1 does not need to be thread-safe as only one thread will push packets into it. The output #2 will only need 3 hardware queues and not 6 (which is the number of threads used on this sample system) as the thread vector shows that only 3 threads can push packets in this element. If not enough queues are available, the thread vector allows to automatically find if the output element need to share one queue among some threads and therefore need to lock before accessing the output ring.

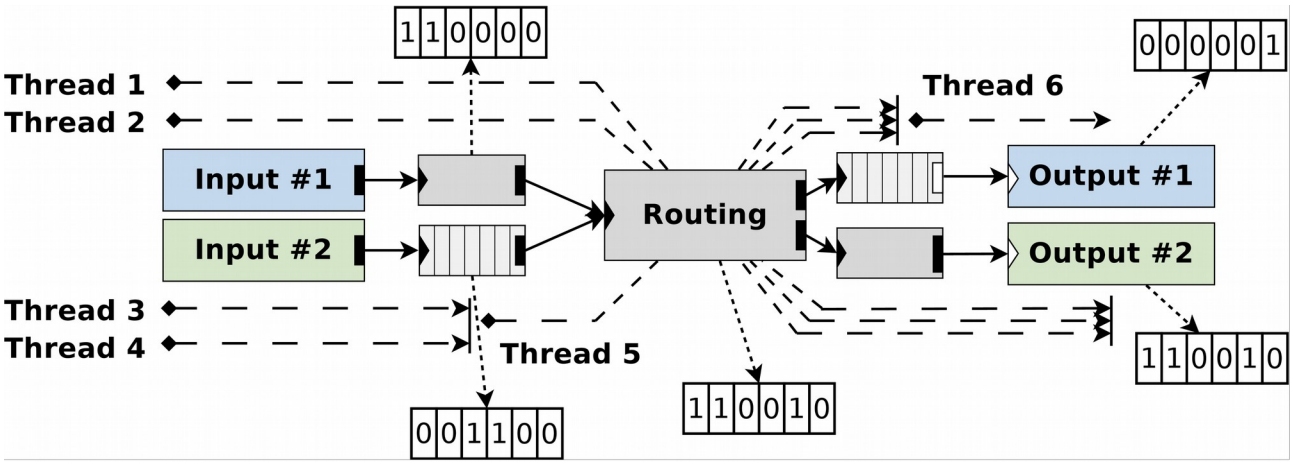


Figure 13: Thread bit vectors used to know which thread can pass through which elements.

Additionally to returning a vector, the function getThreads() can stop the router traversal if the element does not care of its input threads, such as for the queues elements. If that happens, we know that we are not in the full push mode and we'll have to use atomic operations to update the reference counter of the packets as explained in section 2.5.3.

We also provide a per_thread template using the thread vector to duplicate any structure for the thread which can traverse an element to make the implementation of thread-safe elements much more easier. Our model has the advantages of figure 12 (a) and (c) while hiding the multi-queue and thread management and the simplicity of solution (b).

2.5.7 Compute Batching

While compute batching is a well-known concept [11][23], we revisit it in the context of its association with other mechanisms. Moreover, its implementations can differ in some ways.


With compute batching, batches of packets are passed between Click elements instead of only one packet at a time, and the element's job is done on all the packets before transmitting the batch further. SNAP and DoubleClick both use fixed-size batches, using techniques like tagging to discard packets which need to be dropped, and allocating an array for each output of a routing element as big as the input one would, leading to partially-filled batches. We prefer to use a simply linked lists, for which support is already inside Click, and accommodate better with splitting and variable size batches. Packets can have some annotations, and there is an available annotation for a "next" Packet and a "previous" Packet used by the Click packet pool and the queuing elements to store the packets without the need of another data structure. As such, we introduce no new Packet class in Click.

For efficiency we added a "count" annotation which is set on the first packet of the batch to remember the batch size. The "previous" annotation of the first packet is set to the last packet of the batch, allowing to merge batches very efficiently.

The size of the batch is determined by the number of packets available in the receive ring. The batch which comes out of the FromDevice element is composed of all the available packets in the queue, thus only limited by the chosen I/O batching limit.

Compute batching simplifies the output element. The problem with full-push was that a certain number of packets had to be queued before calling the ioctl to flush the output in the Netmap case, or DPDK's function to transmit multiple packets that we'll both refer as the output operation. With compute batching, a batch of packets is received in the output element and the output operation is always done at the end of the sent batch. If the input rate goes down, the batch size will be smaller and the output operation will be done more often, reducing latency as packets don't need to be accumulated.

Without batching, a rate-limit mechanism had to be implemented when the ring is full to limit the call to the output operation. Indeed, in this case, the output operation tends to be called for every packet to be sent, in an attempt to recover space in the output ring. These calls of the output operation can create congestion on the PCIe, a situation to be avoided when many packets need to be sent. This problem naturally disappears when compute batching is used as the time to process the batch gives time to empty part of the output ring.

In both SNAP and DoubleClick, the batches are totally incompatible with the existing Click elements, and you need to use either a kind of

Batcher/Debatcher element (SNAP) or implement new compatible elements. In our implementation, elements which can take advantage of batching inherit from the class BatchElement (which inherit from the common ancestor of all elements “Element”). Before starting the router, Click makes a router traversal, and find BatchElements interleaved by simple Elements. In that case the port between the last BatchElement and the Element will unbatch the packets and the port after the last Element will re-batch them. As the port after the Element cannot know when a batch is finished, the first port calls start_batch() on the downstream port and calls end_batch() when it has finished unbatching all packets. When the downstream port receives the end_batch() call, it passes the reconstructed Batch to their output BatchElement. Note that the downstream port use a per-thread structure to remember the current batch, as multiple batches could traverse the same element at the same time but on different threads.

When a simple (non-batching) element has multiple output, we apply the same scheme but we have to call the start_batch() and end_batch() on all possible directly reachable BatchElement as shown in figure 14. This list is also found on configuration time.

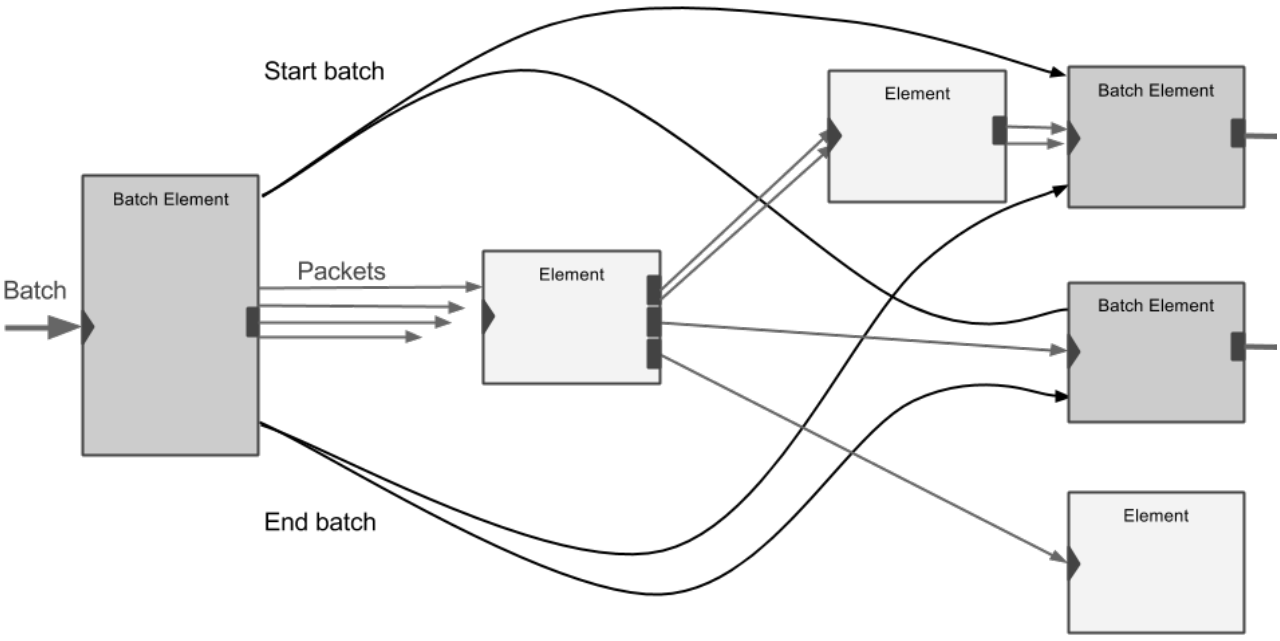


Figure 14: Un-batching and re-batching of packets when downstream elements have multiple paths.

For elements in the fast path, it’s important to implement a support for batching as the cost of un-batching and re-batching would be too big. But for elements in rarely used path such as ICMPError or ARP elements, the cost is


generally acceptable and preferable over development time of elements taking full advantage of compute batching.

Click keeps two pools of objects: one with Packet objects, and one with packet buffers. The previous version of Click and SNAP way of handling a freshly available Netmap packet is to take a packet from the Packet object pool and attach to the filled Netmap buffer from the receive ring. A Netmap buffer from a third pool of free Netmap buffers is then placed back in the receive ring for further reception of a new packet. SNAP does the buffer switching only if the ring is starting to fill up, maintaining multiple type of packets in the pipeline which can return to its original ring or to the third pool; introducing some management cost. We found that it was quicker to have only Netmap buffers in the buffer pool and completely get rid of Click buffer if Click is compiled with Netmap support as the allocate/free of Netmap buffer is done very often. It is very likely that if Netmap is used, packets will be either received or sent from/to a Netmap device. This is called the Netmap pool and is labeled “NPOOL” in figure 15.

Even if Netmap devices are not used, if there is not enough buffers in the pool, the pool can be expanded by calling the same ioctl than in section 2.5.4 to receive a batch of new Netmap buffers and allocate the corresponding amount of Packets. Moreover, our pool is compatible with batching and using the linked list of the batches, we can put a whole batch in the pool at once as we have only one kind of packets, this is called per-batch recycling and is labeled “RECYC” in figure 15.

We do not provide the same functionality in our DPDK implementation. As DPDK always swaps the buffer with a free buffer from its mbuf pool when it receives a packet, we do not need to do it ourself. We simply use the Click pool to take Packet objects and assign them a DPDK buffer.

The results of the router experiment with and without batching for both DPDK and Netmap implementations are shown in figure 15. The “BATCH” label means that the corresponding line uses batching. The “PFTCH” label means that the first cacheline of the packet is prefetched into the CPU’s cache directly when it is received in the input elements. When a packet reaches the “Classifier” element which determine the packet type by reading its Ethernet type field and dispatch packets to different Click paths, the data is already in the cache thanks to prefetching, allowing another small improvement. We omit the forwarding test case because the batching couldn’t improve the performance as it doesn’t do any processing.

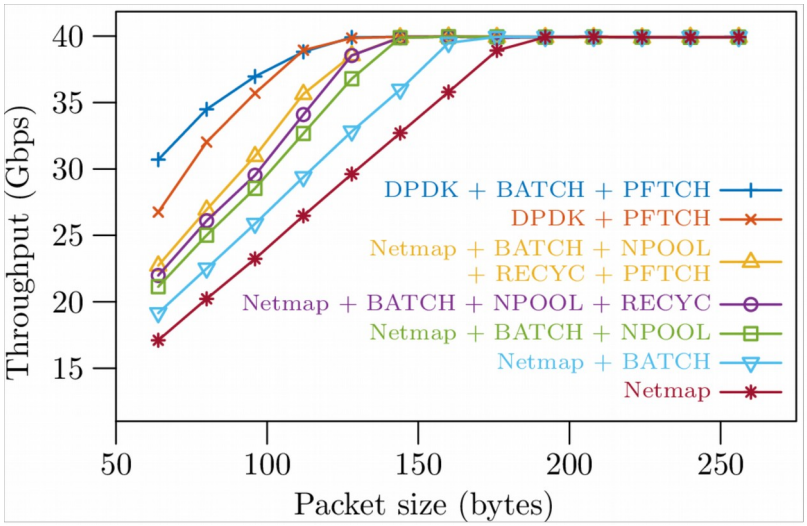


Figure 15: Batching evaluation - DPDK and Netmap implementations with 4 cores using the router test case. See section 2.5.7 for more information about acronyms in legend.

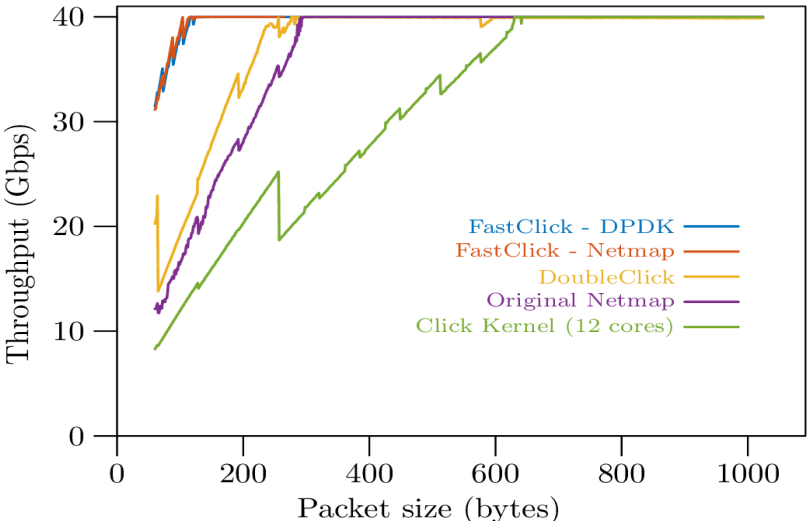


Figure 16: Comparison of forwarding throughput for FastClick and best state-of-the-art Click I/O implementations (using 4 cores except for in-kernel Click).

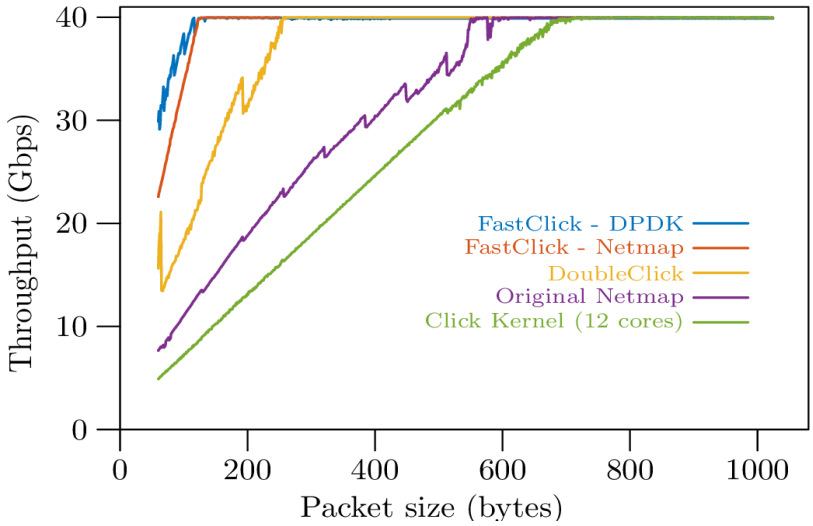


Figure 17: Comparison of router throughput for FastClick and best state-of-the-art Click I/O implementations (using 4 cores except for in-kernel Click).

2.6 FastClick evaluation

We repeated the experiments in section 2.4 with our FastClick implementation. The results of the forwarding experiments are shown in figure 16, and those of the routing experiment in figure 17.

Both Netmap and DPDK FastClick implementations remove the important overhead for small packets, mostly by removing the Click wiring cost by using batches and reducing the cost of the packet pool, using I/O batching and the cost of the packet copy compared to vanilla Click.

The figures do not show an important part of the novelty which is also in the configuration, which becomes much simpler (from 1500 words to 500, without any copy-paste), auto-configured according to NUMA nodes and available CPUs, and the compatibility of pipeline elements with batches.

2.7 Conclusion

We have carried out an extensive study of the integration of packet processing mechanisms and userspace packet I/O frameworks into the Click modular router.

The deeper insights gained through this study allowed us to modify Click to enhance its performance. The resulting FastClick system is backward compatible with vanilla Click elements and was shown to be fit for purpose as a high speed userspace packet processor. It integrates two new sets of I/O elements supporting Netmap and Intel's DPDK.


The context of our work is network middleboxes: packets are received, maybe dropped or modified and then sent through another interface. It may be less suited to scenarios where data is created from scratch in userspace and sent out (as in the case of a server scheme where requests are received as a small number of packets and generate a much bigger number of packets at the output).

Beyond improved performance, FastClick also boasts improved abstractions for packet processing, as well as improved automated instantiation capabilities on modern commodity systems, which greatly simplifies the configuration of efficient Click packet processors. The automatic resource allocation strategies built into FastClick thus seem a promising first step toward the allocation of reusable function blocks (here under the form of Click elements) pipelines onto the available resources (here, under the form of hardware machine nodes).

As many middleboxes operate on the notion of micro-flows, we will extend, as future work, the flow capabilities of FastClick to facilitate the development of such middleboxes.

We will also need to do a similar characterization and optimization in the context of a virtualized infrastructure, *e.g.* running on ClickOS images instead of bare-metal machines.


3 SplitBox: Toward Efficient Private Network Function Virtualization

SplitBox is a scalable system for privately processing network functions in the cloud, that is here used as an example application to assess the suitability of FastClick (see section 2) for middlebox applications. This section only briefly presents SplitBox, and focuses on the evaluation results. For a more comprehensive discussion of the cryptographic algorithms behind SplitBox, see [24].

3.1 Introduction

Network function virtualization (NFV) is increasingly being adopted by organizations worldwide, moving network functions traditionally implemented on hardware middleboxes (MBs) – e.g., firewalls, NAT, intrusion detection systems – to flexible and easier to maintain software processes. Network functions can thus be executed on virtual machines (VMs), with cloud providers processing traffic destined to, or originating from, an enterprise network (the client) based on a set of policies governing the network functions. This, however, implies that confidential information as well as sensitive network policies (e.g., the firewall rules) are revealed to the cloud, whereas in the traditional setting, such policies would only be known to the client's network administrators. Disclosing such policies can reveal sensitive details such as the IP addresses of hosts, the topology of the client's private network, and/or important practices [30][34].

This motivates the need to allow processing outsourced network functions without revealing the policies: we denote this problem as *Private Network Function Virtualization* (PNFV), as done in [32]. We argue that PNFV solutions should not only provide strong *security* guarantees, but also satisfy *compatibility* with existing infrastructures (e.g., not requiring third parties, sending/receiving traffic, take part in new protocols) as well as *high throughput* in order to match the quality of service expected of network functions. In practice, this precludes the use of some standard cryptographic tools as well as other approaches which we review in Section 3.2.

Several attempts have recently been made to support PNFV or similar functionalities [30][31][32][34], assuming the cloud to be honest-but-curious (i.e., the cloud processes the network functions as instructed but may try to learn the underlying policies). However, none of these simultaneously achieve


security, compatibility, and high throughput, or their coverage of network functions is limited as they are only applicable to firewall rules that either allow or drop a packet.

Our intuition is to leverage the distributed nature of cloud VMs: rather than assuming that a single VM processes a client's network function, we distribute the functionality to several VMs residing on multiple clouds or multiple compute nodes in the same cloud. Assuming that not all VMs in the cloud are simultaneously under the control of the adversary (for instance, a *passive* attacker cannot gain access to all nodes running the distributed VMs), we are able to provide a scalable and secure solution. As discussed throughout the paper, achieving this solution is not straightforward and, in the process, we overcome several challenges.

We start by presenting an abstract definition of a network function. Then, we introduce a novel system, which we name SplitBox, geared to privately and efficiently compute this abstract network function in such a way that the cloud, comprising of several middleboxes implemented as VMs, cannot learn the policies. Finally, we implement and evaluate SplitBox on a firewall test case, showing that it can achieve a throughput of over 2 Gbps with 1 kB-sized packets, on average, traversing up to 60 rules.

3.2 Related Work

Khakpour and Liu [30] present a scheme based on Bloom Filters (BFs) to privately outsource firewalls. Besides only considering one use case, their solution is not provably secure as BFs are not *one-way*. Privately outsourcing firewalls is also considered by Shi et al. [34], who rely on CLT multilinear maps [28], which have been shown to be insecure [27]. Jagadeesan et al. [29] introduce a secure multi controller architecture for SDNs based on secure multi-party computation, which can potentially be employed for NFV. However, their implementation takes more than 13 minutes to execute with 4096 flow table entries. Melis et al. [32] investigate the feasibility of provably-secure PNFV for generic network functions: they introduce two constructions based on fully homomorphic encryption and public-key encryption with keyword search (PEKS) [26], however, with high computational and communication overhead (e.g., it takes at least 250ms in their experiments to process 10 firewall rules) which makes it unfeasible for real-world deployment.

Blindbox [33] considers a setting in which a sender (S) and a receiver (R) communicate via HTTPS through a middlebox (MB) which has a set of rules for packet inspection that only it knows. The MB should not be able to decrypt

traffic between S and R, while S and R should not learn the rules. Although Blindbox achieves a 166 Mbps throughput, it operates in a different setting than ours, in which R should set and know the rules (policies), while S and MB should not. Furthermore, the HTTPS connection setup requires around 1.5 minutes with thousands of rules, which suggests that BlindBox may not be practical for applications with short-lived connections.

Finally, Embark [31] enables a cloud provider to support middlebox outsourcing, such as firewalls and NATs, while maintaining confidentiality of an enterprise's network packets and policies. Specifically, it uses symmetric-key encryption to allow communication between enterprises and third-parties or enterprise-to-enterprise. A key difference between Embark and our solution is that we allow complex actions (besides allow/block) to be performed on the packet without revealing them to the cloud, e.g., NAT rules, while Embark can only do so in the clear.

3.3 Preliminaries

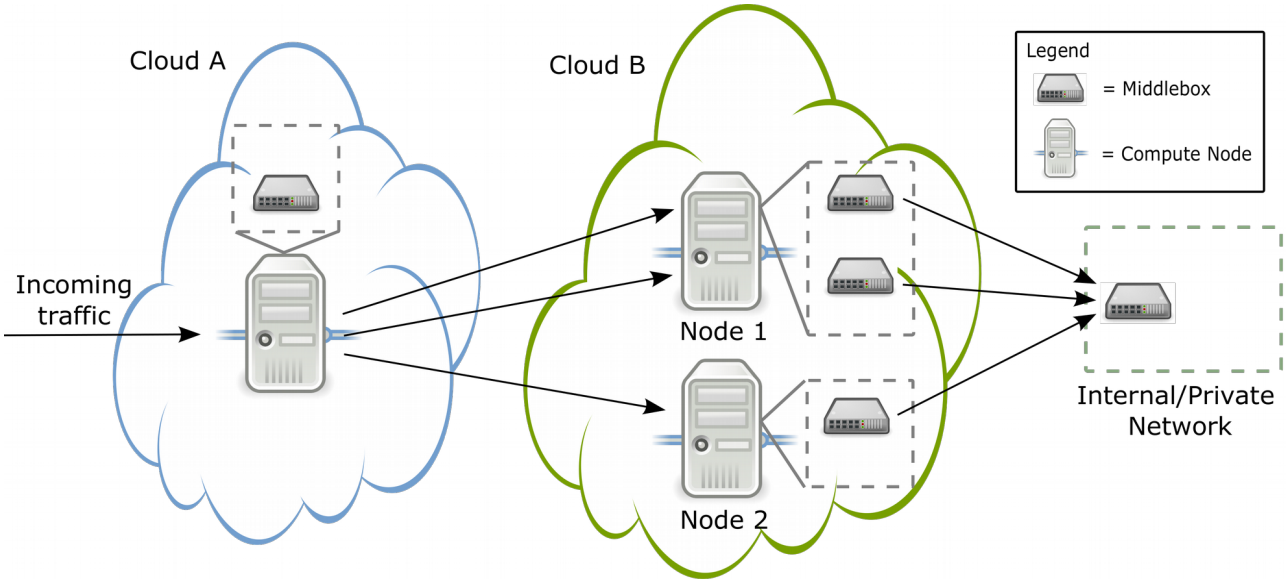


Figure 18: Our system model with Cloud A hosting MB A as a VM in one of its compute nodes. Cloud B hosts the MBs B(t) with t = 3 as VMs (not all t reside on the same compute node). The client MB C resides at the edge of the client's internal network. A and B(t) collaboratively compute network functions for the client.

System and Trust Model. Figure 18 illustrates our PNFV model, consisting of two types of cloud middleboxes (MBs): an *entry* MB $\mathscr{A}$ and $t \geq 2$ cloud MBs $\mathscr{B}(t)$, which collaboratively compute a network function on behalf of a client. The client has its own MB, denoted $\mathscr{C}$, at the edge of its internal network. $\mathscr{A}$ receives an incoming packet, does some computations on it, "splits" the result into t parts, and forwards part j to $\mathscr{B}_j \in \mathscr{B}(t)$. $\mathscr{B}_j$ performs local


computations and forwards its part to $\mathscr{C}$, which reconstructs the network function's final result.

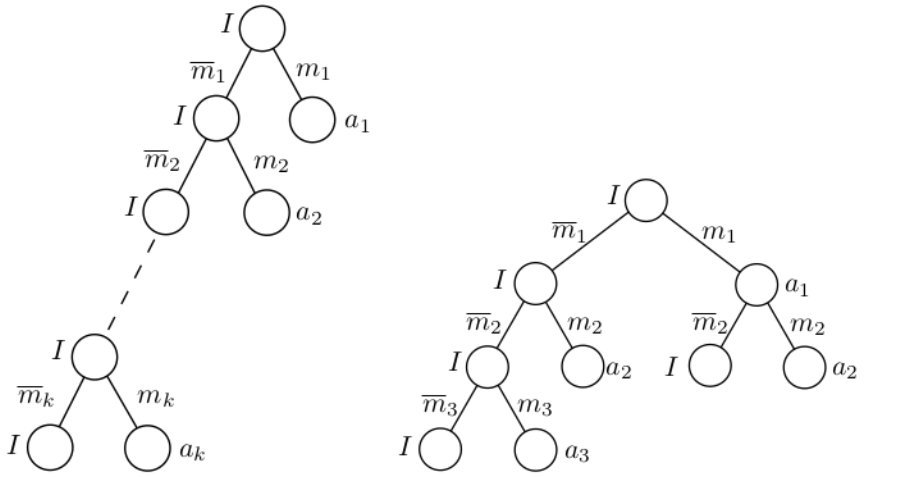
Assumptions. We assume an honest-but-curious adversary which can corrupt[3] either $\mathscr{A}$ or up to $t-1$ MBs from $\mathscr{B}(t)$, and it cannot corrupt $\mathscr{A}$ and any MB in $\mathscr{B}(t)$ simultaneously. In practice, one can assume $\mathscr{A}$ to be running on a different cloud provider than $\mathscr{B}(t)$ and that not all MBs in $\mathscr{B}(t)$ reside on the same node.

Network Functions. We define a packet x as a binary string of arbitrary length. Our network functions will be applicable to the first n bits of x. A *matching* function is a boolean function $m:\{0,1\}^n \rightarrow \{0,1\}$. Its complement, i.e., the function $1-m$, is denoted by $\bar{m}$. An *action* function is a transformation $a:\{0,1\}^n \rightarrow \{0,1\}^n$. $m(x)$ (resp., $a(x)$) denote evaluating m (resp., a) on the substring $x(1,n)$ (i.e., the first n bits of x). If $|x|>n$, a keeps the part $x(n+1,*)$ of x unaltered. We also define the identity action function $I(x)=x$.

Let M and A be finite sets of matching and action functions, with $I \in A$. A *network* function $\psi=(M,A)$ is a binary tree with edge set M and node set A such that each node is an action function $a \in A$ and each edge is either a matching function $m \in M$ or a complement $\bar{m}$ of a matching function $m \in M$. A node is either a leaf node or a parent node. A parent node has two child nodes. The left child node is the identity action function I. The edge connecting the right child node is a matching function $m \in M$, whereas the edge connecting the left child node is its complement $\bar{m}$. The root node is the identity action function I. Clearly, there exists a binary relation from M to A, such that for each (m,a) from this relation there exists a parent node in ψ such that the left child is connected via the edge $\bar{m}$ and the right child via the edge m, and the right child is a.

We call each pair (m,a) in ψ a *policy*. Policies serve as building blocks of a network function. The set of policies of ψ is the set of *distinct* policies (m,a) in ψ. A network function is evaluated on input x, denoted $\psi(x)$, using Algorithm 1. Figure 19 (a) shows a network function with k distinct policies: whenever a match is found, the corresponding action is performed and the function terminates. The function in Figure 19 (b) has 3 distinct policies, $(m_1,a_1),(m_2,a_2)$ and (m_3,a_3), and (m_2,a_2) is repeated twice. This function does not terminate immediately after a match has been found (e.g., path $m_1 m_2$). Since $a \circ I = I \circ a = a$, we can easily "plug" individual policy trees to construct more complex network functions.

3 The adversary may change the behavior of a MB from honest to honest-but-curious.



(a) Network function with k distinct policies.

(b) Network function with 3 distinct policies. Policy (m_2, a_2) is repeated twice.

Figure 19: Network functions as binary trees.

Algorithm 1: `Traversal`

Input: Packet x, network function ψ.

1 Make a read-only copy $x_{\mathtt{r}}$ and a writeable copy $x_{\mathtt{w}}$ of x.
2 Start from the root node.
3 Compute $x_{\mathtt{w}} \leftarrow a(x_{\mathtt{w}})$, where a is the current node.
4 **if** *the current node is a leaf node* **then**
5 | output $x_{\mathtt{w}}$ and stop.
6 **else**
7 | Compute $m(x_{\mathtt{r}})$, where m is the right hand side edge.
8 | **if** $m(x_{\mathtt{r}}) = 1$ **then**
9 | | Move to the right child node.
10 | **else**
11 | | Move to the left child node.
12 Go to step 3.

Figure 20: Private traversal algorithm.

Coverage. Our abstract definition of network functions captures many network functions used in practice. These include firewalls, NAT and load balancers. Such functions usually perform a matching step to inspect some parts of a packet and modify contents of the packet subsequently. In the case of firewalls, modifications may also include dropping a packet.

Branching and chaining. Our definitions support branching, i.e., network functions that do not necessarily apply all policies on a packet. This is achieved by including multiple exit points, i.e., leaf nodes. Definitions also support *chaining*, e.g., ψ_1's output is ψ_2's input, however, in our proposed privacy-preserving solution chaining is not possible, since outputs of the MBs in $\mathscr{B}(t)$ need to be combined to reconstruct a transformed packet.


Policies. We restrict m to substring matching and a to be substring substitution. We also introduce the *don't care bit* denoted by * in our alphabet. Given strings $x\in\{0,1\}^n$ and $y\in\{0,1,*\}^n$, we say $x=y$ if $x(i)=y(i)$ for all $i\in[n]$ such that $y(i)\neq *$. Given $x\in\{0,1\}^*$, matching function m is defined as where $\mu\in\{0,1,*\}^n$. We call μ the *match* of m. Given $x\in\{0,1\}^n$ and $z\in\{0,1,*\}^n$, $x\leftarrow z$ represents replacing each $x(i)$ with $z(i)$ if $z(i)\neq *$, and leaving $x(i)$ as is if $z(i)=*$, for all $i\in[n]$. Given $x\in\{0,1\}^*$, the action function a is defined as where $\alpha\in\{0,1,*\}^n$. We call α the *action* of a.

Definitions. Throughout the rest of this chapter, we use the following definitions: let $z\in\{0,1,*\}^n$, the *projection* of z, denoted π_z, is a string $\in\{0,1\}^n$, such that $\pi_z(i)=1$ if $z(i)\in\{0,1\}$ and $\pi_z(i)=0$ if $z(i)=*$. The *masking* of a $x\in\{0,1\}^n$ using $\pi_z\in\{0,1\}^n$, denoted $\omega(\pi_z,x)$, returns x' such that $x'(i)=x(i)$ if $\pi_z(i)=1$ and $x'(i)=0$ if $\pi_z(i)=0$. $\mathbb{H}:\{0,1\}^n\rightarrow\{0,1\}^q$ denotes a cryptographic hash function; $\oplus$ denotes bitwise XOR. The Hamming weight of a string $x\in\{0,1\}^n$ is $\mathrm{wt}(x)$. Finally, $x\leftarrow_{\$}\{0,1\}^n$ means sampling a binary string of length n uniformly at random.

3.4 Introducing SplitBox

Privacy Requirements. We start by describing an *ideal* setting in which a trusted third party, $\mathscr{T}$, computes a network function ψ for the client. Upon receiving a packet x, $\mathscr{A}$ forwards it to $\mathscr{T}$, which provides the result of $\psi(x)$ to $\mathscr{C}$. Here $\mathscr{A}$ learns x but not $\psi(x)$ and $\mathscr{B}(t)$ neither x nor $\psi(x)$. In this section, we introduce our private NFV solution, SplitBox, aiming to simulate this ideal setting. However, we fall slightly short in that the MBs $\mathscr{B}(t)$ learn the projection π_μ and the output $m(x)$ for each $m\in M$, however, they do not learn the match μ for any $m\in M$ beyond what is learnable from π_μ. Although this could reveal information such as which field of the packet the current matching function corresponds to, we do not consider it to be a strong limitation since this might be obvious from the type of NFV considered anyway. For example, if it is a firewall, then it is common knowledge that the fields it operates on will include IP address fields.

Design Aims. We consider the following design aims, i.e., the solution should: (a) be secure; (b) be computationally fast; (c) limit MB-to-MB communication complexity.

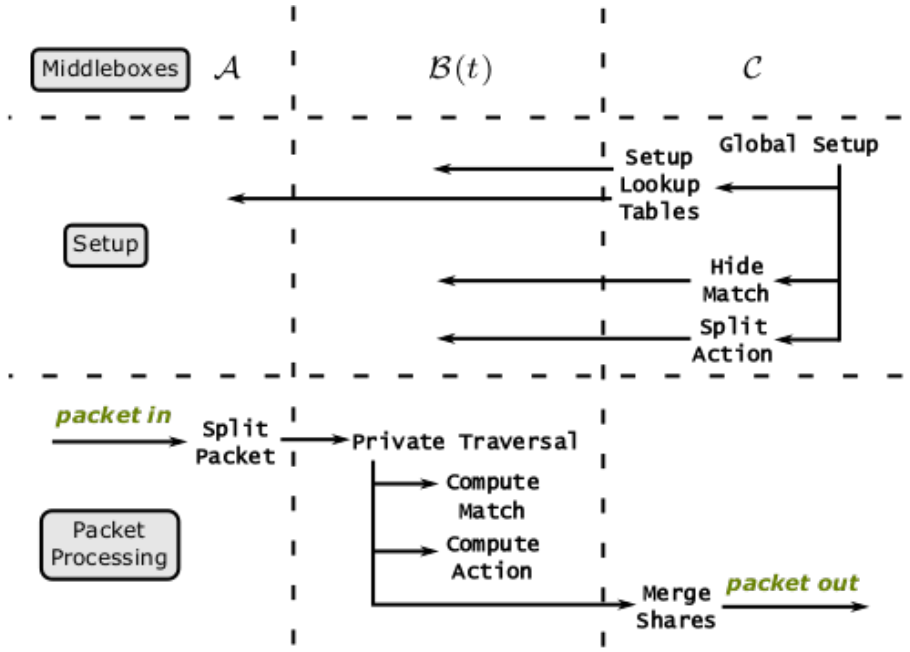


Figure 21: Breakdown of algorithms executed by each MB in SplitBox.

High-Level Overview. In a nutshell, if we assume that ψ includes a single policy (m,a), our strategy to hide m is to let $\mathscr{C}$ *blind* μ by XORing it with a random binary string s and sending the hash of the result to each MB in $\mathscr{B}(t)$; whereas, to hide a, $\mathscr{C}$ computes t *shares* of the action α using a t-out-of-t secret sharing scheme and sends share j to $\mathscr{B}_j$. In addition, $\mathscr{A}$ encrypts the contents of a packet x by XORing it with the blind s, and sends it to the MBs in $\mathscr{B}(t)$, which can then compute matches and actions on this encrypted packet. We present the details of SplitBox using a set of algorithms, grouped based on the MB executing them. Figure 21 shows a high-level overview of all the algorithms computed by each MB. We assume ψ_{priv} to be the private version of the network function ψ whose matching and action functions are replaced by unique identifiers.

Middlebox $\mathscr{C}$. The initial setup is performed by $\mathscr{C}$ via Algorithm `Global Setup` (figure 22). This includes creating lookup tables via Algorithm `Setup Lookup Tables` (figure 23), hiding the matching functions via Algorithm `Hide Match` (figure 24), and splitting the action functions via Algorithm `Split Action` (figure 25). There are two lookup tables in Algorithm 3: S for $\mathscr{A}$ and $\widetilde{S}$ for $\mathscr{B}(t)$. Table S contains l "blinds" which are random binary strings used to encrypt a packet by XORing. For each blind $s \in S$ and for each $m \in M$, the portion of the blind corresponding to the projection of the match μ is extracted and then XORed with μ. Finally this value is hashed using $\mathbb{H}$ and stored in the corresponding row of $\widetilde{S}$. The `Hide Match` algorithm simply sends the


projection π_μ of each match μ to $\mathscr{B}(t)$. This tells $\mathscr{B}(t)$ which locations of the incoming packet are relevant for the current match. The `Split Action` algorithm computes t shares of the action α and action projection π_α, for each $a \in A$ and sends them to $\mathscr{B}(t)$. $\mathscr{C}$ uses one more algorithm, Algorithm `Merge Shares` (figure 26) to reconstruct the transformed packet. This algorithm XORs the cumulative action shares α'_j and cumulative action projection shares β'_j from $\mathscr{B}_j$ to compute the final action α' and action projection β'. It also XORs the encrypted packet received from $\mathscr{A}$ with the current blind s in the lookup table S, in order to reconstruct the final packet. Note that we have modeled dropping a packet as setting $x(1,n)$ to 0^n.

Algorithm 2: `Global Setup` $(\mathcal{C})$

Input: Parameters n and l, network function $\psi = (M, A)$.

1 **for** $j = 1$ **to** t **do**
2 | Send ψ_{priv} to $\mathcal{B}_j$.
3 Run `Setup Lookup Tables` with parameter l, M.
4 **for** *each* $m \in M$ **do**
5 | Run `Hide Match` algorithm.
6 **for** *each* $a \in A$ **do**
7 | Run `Split Action` algorithm.

Figure 22: Global Setup algorithm.

Algorithm 3: `Setup Lookup Tables` $(\mathcal{C})$

Input: Parameter l, set M.

1 Initialize empty table S with l cells.
2 Initialize empty table $\tilde{S}$ with $l \times |M|$ cells.
3 **for** $i = 1$ **to** l **do**
4 | Sample $s_i \leftarrow_\$ \{0,1\}^n$.
5 | Insert s_i in cell i of S.
6 | **for** $j = 1$ **to** $|M|$ **do**
7 | | Compute $\tilde{s}_{i,j} = \omega(\pi_{\mu_j}, s_i)$, where μ_j is the match of m_j.
8 | | Compute $\mathbb{H}(\mu_j \oplus \tilde{s}_{i,j})$.
9 | | Insert $\mathbb{H}(\mu_j \oplus \tilde{s}_{i,j})$ in cell (i,j) of $\tilde{S}$.
10 Send S to $\mathcal{A}$.
11 Send $\tilde{S}$ to $\mathcal{B}(t)$.

Figure 23: Setup Lookup Tables algorithm.

Algorithm 4: `Hide Match` $(\mathcal{C})$

Input: Matching function $m \in M$ with match μ.

1 Send π_μ to $\mathcal{B}(t)$.

Figure 24: Hide Match algorithm.


Algorithm 5: `Split Action` ($\mathcal{C}$)

Input: Action function $a \in A$ with action α.

Sample $\alpha_1, \alpha_2, \ldots, \alpha_{t-1} \leftarrow_{\$} \{0,1\}^n$.
Let $\tilde{\alpha} = \omega(\pi_\alpha, \alpha)$. Compute $\alpha_t = \tilde{\alpha} \oplus \alpha_1 \oplus \cdots \oplus \alpha_{t-1}$.
Sample $\beta_1, \beta_2, \ldots, \beta_{t-1} \leftarrow_{\$} \{0,1\}^n$.
Compute $\beta_t = \pi_\alpha \oplus \beta_1 \oplus \cdots \oplus \beta_{t-1}$.
for $j = 1$ **to** t **do**
    Give α_j, β_j to $\mathcal{B}_j$.

Figure 25: Split Action algorithm.

Algorithm 6: `Merge Shares` ($\mathcal{C}$)

Input: Index i, packet copy $x_{\mathtt{w}}$, α'_j and β'_j from $\mathcal{B}_j$ for $j \in [t]$.

Compute $\alpha' \leftarrow \alpha'_1 \oplus \cdots \oplus \alpha'_t$.
Compute $\beta' \leftarrow \beta'_1 \oplus \cdots \oplus \beta'_t$.
Compute $x \leftarrow x_{\mathtt{w}} \oplus s_i$, where $s_i \in S$.
for $i = 1$ **to** n **do**
    if $\beta'(i) = 1$ **then**
        $x(i) \leftarrow \alpha'(i)$
if $x(1,n) = 0^n$ **then**
    Drop x.
else
    Forward x.

Figure 26: Merge Shares algorithm.

Middlebox $\mathscr{A}$. This MB only runs Algorithm `Split Packet` (figure 27), which maintains a counter initially set to 0 and incremented every time a new packet x arrives. The value of the counter corresponds to a blind in the lookup table S. Therefore its range is $[l]$ (barring the initial value of 0). The algorithm makes two copies of an incoming packet x, $x_{\mathtt{r}}$ (read-only copy) for matching to be sent to $\mathscr{B}(t)$, and $x_{\mathtt{w}}$ (writeable copy) for action functions to be sent to $\mathscr{C}$. Both $x_{\mathtt{r}}$ and $x_{\mathtt{w}}$ are XORed with the blind in S corresponding to the counter. The current counter value is also given to $\mathscr{B}(t)$ and $\mathscr{C}$.

Algorithm 7: `Split Packet` ($\mathcal{A}$)

Input: Packet x, lookup table S.

Get the index $i \in [l]$ corresponding to the current value of the counter.
Let $x_{\mathtt{w}} \leftarrow x \oplus s_i$ (writeable copy), where $s_i \in S$.
Compute $x_{\mathtt{r}} \leftarrow x(1,n) \oplus s_i$ (read-only copy), where $s_i \in S$.
for $j = 1$ **to** t **do**
    Send $x_{\mathtt{r}}, i$ to $\mathcal{B}_j$.
Send $x_{\mathtt{w}}, i$ to $\mathcal{C}$.

Figure 27: Split Packet algorithm.

Middleboxes $\mathscr{B}(t)$**.** Each MB $\mathscr{B}_j$ performs a private version of the `Traversal` algorithm as shown in Algorithm `Private Traversal` (figure 28). $\mathscr{B}_j$ first initializes cumulative action strings α'_j and cumulative action projection strings β'_j as strings of all zeros. Within the `Private Traversal` algorithm, $\mathscr{B}_j$ executes the action functions using Algorithm `Compute Action` (figure 29) and matching functions using Algorithm `Compute Match` (figure 30). The `Compute Action` algorithm essentially updates α'_j and β'_j by XORing with the action share and action projection share of the current action. The `Compute Match` algorithm uses the read-only copy x_r. It extracts the bits of x_r corresponding to the current match projection π_μ. It then looks up the counter value i (sent by $\mathscr{A}$) and the index of the matching function in the lookup table $\widetilde{S}$ and extracts the hashed match. This is then compared with the hash of the relevant bits of x_r.

Algorithm 8: `Private Traversal` $(\mathcal{B}(t))$

Input: Index i, read-only copy $x_{\mathbf{r}}$, network function ψ_{priv}.

1 Initialize empty strings $\alpha'_j \leftarrow 0^n$ and $\beta'_j \leftarrow 0^n$.
2 Start from the root node.
3 Update α'_j and β'_j by running the `Compute Action` algorithm on the current node a.
4 **if** *the current node is a leaf node* **then**
5 Send i, α'_j and β'_j to party $\mathcal{C}$ and stop.
6 **else**
7 Run `Compute Match` algorithm on i, m and $x_{\mathbf{r}}$, where m is the right hand side edge.
8 **if** `Compute Match` *outputs 1* **then**
9 Go to the right child node.
10 **else**
11 Go to the left child node.
12 Go to step 3.

Figure 28: Private Traversal algorithm.

Algorithm 9: `Compute Action` $(\mathcal{B}(t))$

Input: Pair of cumulative action and cumulative action projection shares (α'_j, β'_j) of $\mathcal{B}_j$, pair of action and action projection shares (α_j, β_j) of action function $a \in A$ of $\mathcal{B}_j$.

1 Compute $\alpha'_j \leftarrow \alpha'_j \oplus \alpha_j$.
2 Compute $\beta'_j \leftarrow \beta'_j \oplus \beta_j$.
3 Output α'_j, β'_j.

Figure 29: Compute Action algorithm.

Algorithm 10: `Compute Match` $(\mathcal{B}(t))$

Input: Read-only copy $x_{\mathtt{r}}$, index $i \in [l]$, lookup table $\tilde{S}$, index $j \in [|M|]$ of $m_j \in M$ with match μ_j.

1 Lookup table $\tilde{S}$ at index (i, j) to obtain $\mathbb{H}(\tilde{s}_{i,j})$.
2 Extract $\tilde{x}_{\mathtt{r}} \leftarrow \omega(\pi_{\mu_j}, x_{\mathtt{r}})$.
3 Compute $\mathbb{H}(\tilde{x}_{\mathtt{r}})$.
4 **if** $\mathbb{H}(\tilde{x}_{\mathtt{r}}) = \mathbb{H}(\mu_j \oplus \tilde{s}_{i,j})$ **then** // $m(x) = 1$
5 | Output 1.
6 **else** // $m(x) = 0$
7 | Output 0.

Figure 30: Compute Match algorithm.

3.5 Analysis

Correctness. Given $\psi=(M,A)$, for a matching function $m\in M$, as long as m can be represented as substring matching, SplitBox correctly computes the match. That is, if m is an equality test or range test for powers of 2 in binary (e.g., IP addresses in the range $127.*.*.32$ to $127.*.*.64$), then it can be successfully computed by SplitBox. Our model also allows for arbitrary ranges by dividing m into smaller matches that check equality matching of individual bits. However, such a representation can potentially make ψ very large. We can correctly compute action functions as long as they satisfy two properties: (a) they are applied to the initial packet x only, and not on its transformed versions; (b) any two action projections β_i and β_j do not overlap on their non-zero bits. Note that this does not restrict the number of times the identity function I can be applied, as its action projection is 0^n.

Security. While we refer to [24] for the security proofs, here we mention two important points: if SplitBox is used for match projections whose Hamming weight is low, then the $\mathscr{B}(t)$ can brute-force $\mathbb{H}$ to find its pre-image. This reveals $\mu\oplus s$ for some blind s, which allows the adversary to learn more than simply looking at the output of m. Namely, if $m(x)=0$, the adversary learns which relevant bits of an incoming packet x do not match with the stored match. The second point relates to the length of the look-up table l: ideally l should be large enough so that the same blind is not re-used before a long period of time. However, high throughput would require a prohibitively large value of l. Therefore, we propose the following mitigation strategy: with probability $0<1-\rho<1$, $\mathscr{A}$, sends a uniform random string from $\{0,1\}^n$ (dummy packet), rather than the next packet in the queue.

3.6 Implementation

In this section, we discuss our proof-of-concept implementation of SplitBox inside FastClick [25], an extension of the Click modular router [12] which provides fast user-space packet I/O and easy configuration via automatic handling of multi-threading and multiple hardware queues. We also use Intel DPDK [10] as the underlying packet I/O framework. We implemented three main FastClick elements: element `Entry` corresponding to MB $\mathscr{A}$, `Processor` corresponding to MBs $\mathscr{B}(t)$, and `Client` to $\mathscr{C}$. `Client` implements the `Merge Shares` algorithm. The other algorithms of $\mathscr{C}$ are executed outside the FastClick elements, and used to configure the above three elements. The hash function $\mathbb{H}$ is implemented using OpenSSL's SHA-1, aiming to achieve a compromise between security, digest length, and computation speed, as hash functions which have larger message digests will lead to overly large lookup tables. `Client` uses a circular buffer to collect packet shares until all have been received and the final packet can be reconstructed. For communication between our elements, we use UDP packets: UDP and L2 processing relies on standard Click elements such as `UDPIPEncap`. Finally, we also add a few elements to help in our delay measurements, as explained below.

To evaluate our implementation, we focus on a firewall use case, using a network function tree similar to that in Figure 19 (a). A single action is applied, either the identity action, if the packet is allowed, or marking the packet with a drop message (0^n), if it should be dropped. We use three commodity PCs for our experiments (8-core Intel Xeon E5-2630 with 2.4GHz CPU and 16 GB of RAM): one for both `Entry` and `Client`, in order to use the same clock for delay measurements, and the other two as two `Processor`s. The four nodes (including the two on the same machine) are connected through Intel X520 NICs, with 10-Gbps SFP+ cables. The topology is thus very similar to the one in Figure 18, except that we only have t=2 in $\mathscr{B}(t)$, and that $\mathscr{A}$ and $\mathscr{C}$ share the same physical machine. Another difference is that our machines are connected directly, without intermediate routers between them. We use a trace captured at one of our campus border router (pre-loaded into memory) as input for the `Entry` element, which executes the `Split Packet` algorithm on a single core. Then, each output of `Entry` (one for $\mathscr{C}$ and one per $\mathscr{B}_j$) is encapsulated inside an UDP packet and sent to the corresponding output device, using one core per device.

On each $\mathscr{B}_j$ machine, the packets are read from the input device, decapsulated, and then passed to a `Processor` element which does the actual


filtering. The resulting action packets are then re-encapsulated and sent through the NIC towards the client. This operation is done on a single core, but several cores can easily be used in parallel. With FastClick, it suffices to launch Click with more cores, and the system will automatically create the corresponding number of hardware queues on the NICs, and assign a core to each queue. On the client side, each of the three input NICs has an associated core. Incoming packets are decapsulated, and then passed to the `Client` element, which reconstructs the final packets (on its own core). Reconstructed packets which are not marked as dropped are then passed to a receiver pipeline, which computes the entry-to-exit delay, counts packets and measures reception bandwidth. To measure delays, the packets in the in-memory list are tagged with a sequence number in the packet payload, before the transmission begins. This number allows to match the exit time-stamp with an entry time-stamp, which is kept in memory. This allows to avoid storing the time-stamp itself in the packet, which would increase the delay measured.

3.7 Performance Evaluation

We now present the results of the experiment described above, with various input bit-rates and different number of rules, while measuring loss rate and delays. While we have to forward all packets to the client, a non-private outsourced firewall can drop the rejected packets immediately. Thus, its achievable bit-rate will depend on a combination of the input traffic and the ruleset. To normalize results in our analysis, we craft rulesets such that all packets are accepted. While it changes nothing for SplitBox, it is a worst-case for the `IPFilter`-based test-case. At the same time, we tightly control the number of match attempts per packet, in order to evaluate the impact of the average number of rules traversed by a packet before it matches.

Figure 31 illustrates the evolution of the maximum achievable bandwidth (taken as inducing less than 0.001% losses), as a function of the number of traversed rules (i.e., the number of match attempts per packet). Our trace packets are about 1 kB on average, so that 8 Gbps corresponds to about 1 Mpps. We observe that the bandwidth decreases significantly with more traversed rules with SplitBox (PNFV), mainly due to the hashing function, which is called on the packet header once per match attempt. Not only is this more computationally expensive than simpler comparisons, but it is also done each time on different data (as we need to first XOR packet header with match projection), taking no advantage of the cache. Fortunately, the `Processor` operation is inherently parallelizable, thus, allocating more cores speeds things up. Note that the average number of traversed rules in a real firewall is

significantly lower than the total number of rules. Therefore, it is particularly important to choose the order of match attempts according to the traffic distribution, and/or to use a more complex tree structure minimizing the number of match attempts.

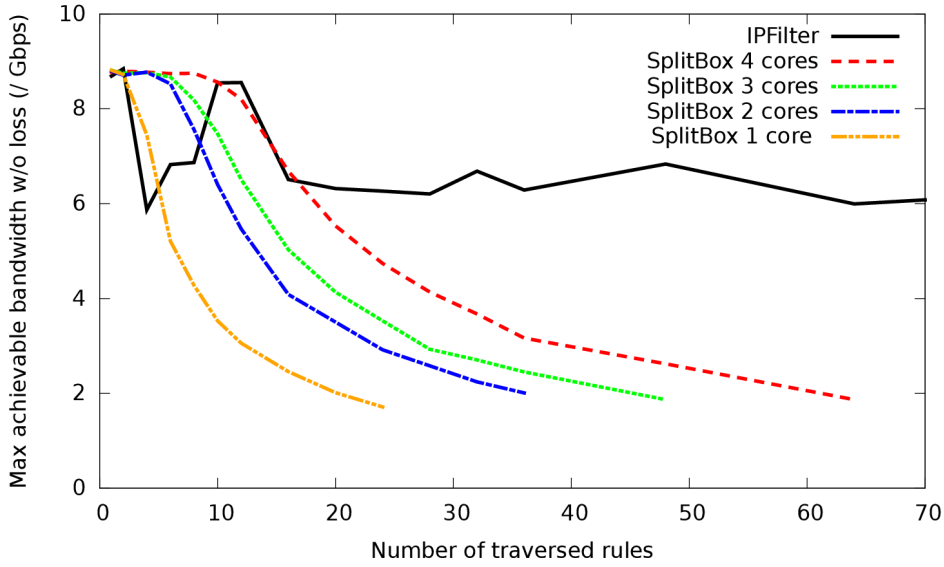


Figure 31: Achievable bandwidth drops sharply with the number of traversed rules.

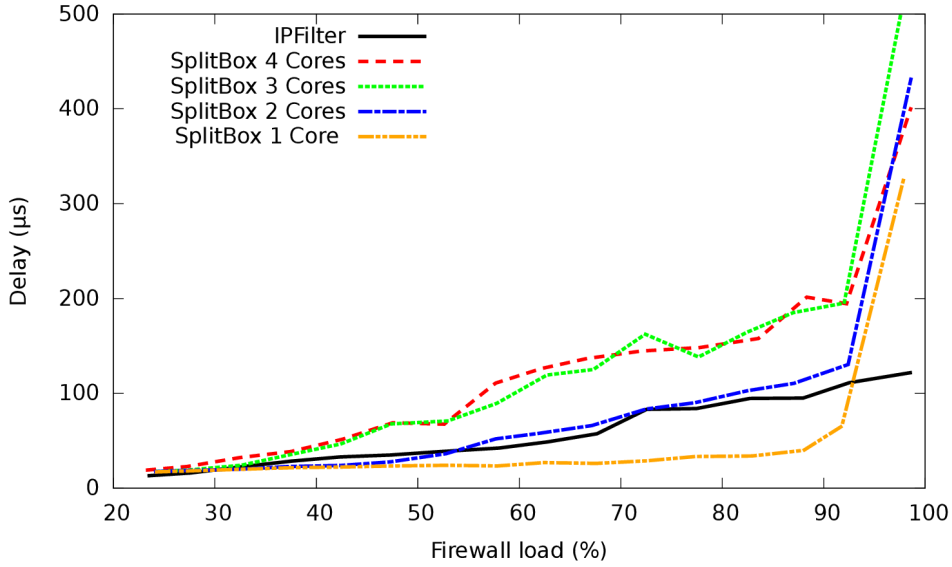


Figure 32: Delay increases with the firewall load.

Finally, in Figure 32, we plot the delays as a function of firewall load (i.e., current input bandwidth over maximum achievable bandwidth). Note that the delays do not follow the same dependency with regard to the number of match attempts per packet. Although these increase slightly with the number of traversed rules, they are mostly governed by queuing delays in the system (in NICs rings, or in-memory rings exchanging packets between the different


processing cores). The number of blinds l seems to have little impact on the performance: with l ranging from 64 to 65,536, we observe no noticeable difference, except for additional memory consumption.

In conclusion, our SplitBox proof-of-concept implementation for a firewall use case achieves comparable performance to a non-private version, providing acceptable throughput and delays for small rulesets. Larger rulesets should be carefully laid out in order to minimize the number of match attempts per packet.

3.8 Conclusion & Future Work

This chapter presented SplitBox, a novel scalable system that allows a cloud service provider to privately compute network functions on behalf of a client, in such a way that the cloud does not learn the network policies. It provides strong security guarantees in the honest-but-curious model, based on cryptographic secret sharing. We experiment with our implementation using firewall as a test case, and achieve a throughput in the order of 2 Gbps, with packets of average size 1 kB traversing about 60 firewall rules.

This shows the suitability of FastClick for the deployment of middlebox functionality in the cloud. FastClick allows the user to concentrate on the implementation of the specific network functionality, and automatically handles low-level allocation details (*e.g.* queue and core allocation), while still providing very good performance.


4 Characterization of an NFV infrastructure under different workloads

Orchestration of workloads in a data center needs to consider at least two key aspects in order to improve life-cycle management and to improve the intelligence of orchestration decisions:

- The amount of resources dedicated to a customer workload:

  The deployment configuration parameters, that include for example the number of CPUs, number of virtual NICs, the amount of RAM, the number of SR-IOV channels etc., are very important in order to guarantee the normal execution of the service and to support any SLA and/or KPI required by the end-user. It is important to measure the amount of resources required by a service in order to measure the nominal capacity that could potentially be used by the service itself.

- The real utilization of the infrastructure resources:

  The nominal capacity required by the service is not always coincident with the real usage of the infrastructure over time by the service. From an Infrastructure/Service provider perspective, it is interesting to understand and measure the impact of a workload execution profile for a given infrastructure environment. This information could be exploited by smart orchestration frameworks to make improved decisions around the scheduling of workloads on the infrastructure, in order to, on the one side, avoid SLA violations according to the specific service KPIs and, on the other side, maximize infrastructure utilization which in turn results in maximization of the return of investment through application of consolidation approaches within the data centers.

In order to cope with both of these aspects and to support the modeling of a service, a fingerprinting approach has been investigated: it is a methodology that allows an orchestrator to access data and statistics about the general behavior of workloads and the effects that they have on the underlying infrastructure.

The general goal of the approach is to couple information about both physical resources involved in the execution of a workload and the telemetry collected in relation to them. This can generate insights about the effect that a given workload has on the physical infrastructure when using different deployment configurations in different scenarios.


The resource model taken into account for this purpose is the infrastructure model described in D4.1 (Heterogeneous Infrastructure – abstraction with resource differentiation). In the context of that model, all the resources included in a data center are categorized into three main domains: compute, network and storage. From a fingerprinting perspective, the research work conducted to date has been focused on these three domains which have been explored with the help of the aforementioned infrastructure model.

The current proposed methodology automatically extracts a graph representation of a workload, including the service layer description, the virtual resources deployed to support the service and the physical resources involved in the execution of them.

The graph is therefore used to collect profiling information about the capacity statically allocated to the service and the real usage of the infrastructure.

Preliminary investigations have been carried out using sample workloads to stress the compute, network and storage components of an NFV infrastructure (NFVI). The following results are related to three different workloads:

- Workload 1 – Network stressing tool based on the Linux *pktgen* [35], composed by two Virtual Machines. One of them generates the traffic, whereas the other receives the traffic generated by the first.
- Workload 2 – CPU stressing tool based on Linux software *stress* [36], composed by one Virtual Machine.
- Workload 3 – Storage stressing tool based on *iozone* [37], composed by one virtual machine writing to and reading from the disk.

Figure 33, 34 and 35 have been automatically generated on the basis of information included in the infrastructure model and the telemetry data collected through an agent deployed on the NFVI nodes.

Figure 33 shows the results of the nominal capacity analysis for the 3 workloads under investigation.

Figure 34 and 35 show the results of the actual utilization of resource. More specifically figure 34 shows the specific utilization of the resources over a given time window, whereas figure 35 shows their standard deviations.

Figure 33: Static capacity allocated to the service.

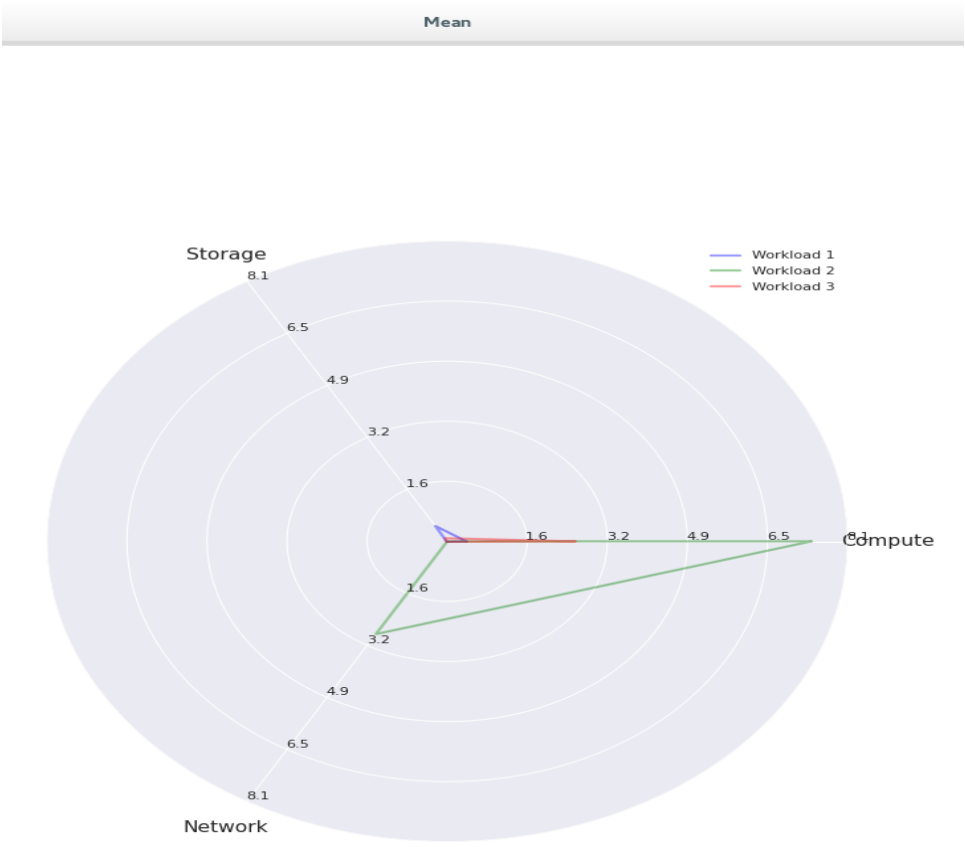


Figure 34: Mean of utilization of compute, network and storage resources in the platform

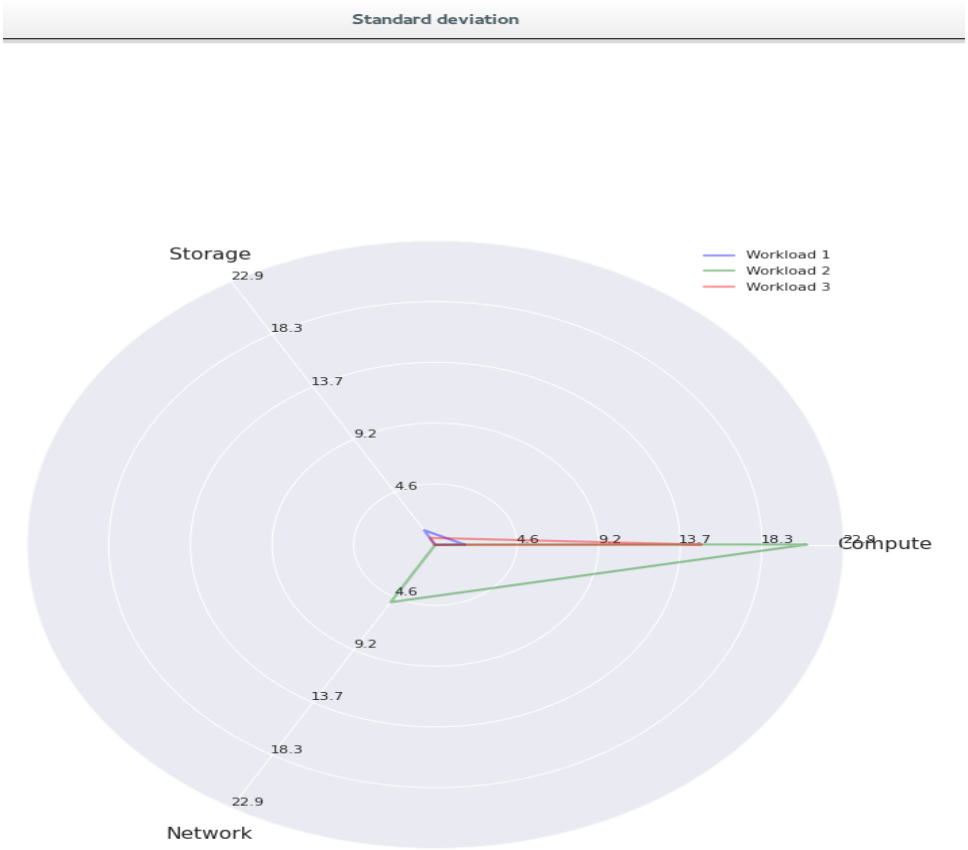


Figure 35: Standard deviation of utilization of compute, network and storage resources in the platform.

The results presented are based on running the workload in isolation on the servers and with fixed deployment configurations. Further development of the fingerprinting approach will include the investigation of different configuration deployments, in order to evaluate how a workload changes behavior in terms of impact on the platform when it changes configuration. Additionally, the exploration of more specific metrics will be investigated as well as the potential combination of the fingerprinting with the output of the KPI mapping activity conducted in task 4.1, in order to exploit this approach for orchestration purposes.


5 Application of performance optimization to the MicroVisor virtualization platform

A number of user-space I/O performance optimizations have been discussed in Section 2. To try and further optimize the performance we have also developed some of the features and optimization techniques in the virtualization layer. The MicroVisor is a distributed, light-weight hypervisor platform that has been designed with performance and stability as two guiding attributes. It is derived from the commonly used Xen platform but removes the control domain (Dom0) and instead distributes the control and command logic within the MicroVisors. The design and implementation of the MicroVisor is not carried out in Superfluidity but the performance optimizations and the adaption of the platform to work with the Superfluidity architecture is being carried out in this project.

The MicroVisor exposes the hardware primitives and the acceleration features of the underlying hardware up through to the higher layers of the platform. This allows features such as NUMA-set pinning to be carried out that have a large impact on the performance of the workloads running on the platform.

The performance optimization features that are available in the MicroVisor platform are not limited to:

- Virtual CPU to Physical CPU pinning.
- CPU pool per NUMA node.
- Memory NUMA awareness through the xen feature numa=on.
- Expose block-level storage devices to the physical network via ATA over Ethernet to accelerate remotely accessible storage (ATA-over-Ethernet block device backend in the MicroVisor layer to allow block requests over Ethernet).
- ATA over Ethernet – kernel module running in storage node VM as a server.
- End-to-end support for virtual network packet fragmentation including TSO/GSO.
- Round-robin over Virtual NIC queues for sending data from the MicroVisor to a VM which is multi-queue capable.


- Pinning of Netfront and Blkfront queues to particular vCPUs within the storage node and driver domain.
- Link aggregation between MicroVisor nodes using the underlying physical network configuration options (striping, load-balancing, high-availability, daisy-chain).
- Virtual network overlays over the available physical network topology.
- Lightweight Network Functions for packet processing, routing, encryption as unikernel instances, able to move across the infrastructure with minimum downtime.

6 Conclusion & Future work

This intermediate version of the deliverable concentrated on the issues related to the allocation and placement of low-level processing modules on the physical resources of a single hardware box (whether bare-metal, or in a virtualized setting). These issues are fundamental to the deployment of virtual network functions and are a necessary stepping stone towards to higher level problem of allocating network functions on an single middlebox, based on network traffic performance SLAs. Beyond this more high-level allocation problem, cloud-based network processing platform management will also require efficient solutions to the problem of allocating high-level network functions across distributed cloud VMs. Another useful step in that direction is the characterization of a virtualized NFV infrastructure under different workloads.

7 References

[1] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy. Can software routers scale? In Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow, PRESTO'08, pages 21-26, New York, NY, USA, 2008. ACM.

[2] ASUS. P9x79-e ws. http://www.asus.com/Motherboards/P9X79E_WS/.

[3] T. Barbette. Click pull request #162 to enable multi-producer single-consumer mode in Linux module FromDevice. https://github.com/kohler/click/pull/162.

[4] T. Barbette. Tom Barbette's research part. http://www.tombarbette.be/research/.

[5] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association.

[6] 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, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[7] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In Proceedings of the 2008 ACM CoNEXT Conference, page 20. ACM, 2008.

[8] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[9] Intel. Core TM i7-4930k processor (12m cache, up to 3.90 ghz). http://ark.intel.com/products/77780.

[10] Intel. Data plane development kit. http://www.dpdk.org.

[11] 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, APSYS '12, pages 14:1–14:6, New York, NY, USA, 2012. ACM.

[12] 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.

[13] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, page 9. ACM, 2013.

[14] Linux Kernel Contributors. Packet mmap. https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt.

[15] ntop. DNA vs netmap. http://www.ntop.org/pf_ring/dna-vs-netmap/.

[16] ntop. PF RING. http://www.ntop.org/products/pf_ring/.

[17] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[18] L. Rizzo. Device polling support for freebsd. In BSDConEurope Conference, 2001.

[19] L. Rizzo. Netmap: A novel framework for fast packet I/O. In USENIX Annual Technical Conference, pages 101-112, 2012.

[20] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet i/o in virtual machines. In Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems, pages 47–58. IEEE Press, 2013.

[21] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet.

[22] Solarflare. OpenOnload. http://www.openonload.org/.

[23] W. Sun and R. Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Oct. 2013.

[24] H. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. Kaafar, and L. Mathy. SplitBox: Toward Efficient Private Network Function Virtualization. https://arxiv.org/abs/1605.03772. 2016.
[25] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2015.
[26] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In Eurocrypt, 2004.
[27] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehle. Cryptanalysis of the Multilinear Map over the Integers. In Eurocrypt, 2015.
[28] J.-S. Coron, T. Lepoint, and M. Tibouchi. Practical Multilinear Maps over the Integers. In CRYPTO, 2013.
[29] N. A. Jagadeesan, R. Pal, K. Nadikuditi, Y. Huang, E. Shi, and M. Yu. A Secure Computation Framework for SDNs. In HotSDN '14, 2014.
[30] A. R. Khakpour and A. X. Liu. First Step Toward Cloud-Based Firewalling. In SRDS, 2012.
[31] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In NSDI, 2016.
[32] L. Melis, H. J. Asghar, E. De Cristofaro, and M. A. Kaafar. Private Processing of Outsourced Network Functions: Feasibility and Constructions. In ACM Workshop on SDN-NFV Security, 2016.
[33] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In SIGCOMM, 2015.
[34] J. Shi, Y. Zhang, and S. Zhong. Privacy-preserving Network Functionality Outsourcing. http://arxiv.org/abs/1502.00389, 2015.
[35] The Linux Foundation. Packetgen. http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen.
[36] A. Waterland. Stress. http://people.seas.harvard.edu/~apw/stress/.
[37] W. Norcott. IOzone. http://www.iozone.org/.
[38] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Forwarding path architecture for multicore software routers. In ACM PRESTO, 2010.