Architecture for programmable network infrastructure

Tom Barbette

Université de Liège

Faculté des Sciences Appliquées

Département d'Electricité, Electronique et Informatique

Doctoral Dissertation presented in fulfilment of the requirements for the degree of

Docteur en Sciences (Informatiques)

May 2018

Summary

Software networking promises a more flexible network infrastructure, poised to leverage the computational power available in datacenters. *Virtual Network Functions (VNF)* can now run on commodity hardware in datacenters instead of using specialized equipment disposed along the network path. VNFs applications like stateful firewalls, carrier-grade NAT or deep packet inspection that are found “in-the-middle”, and therefore often categorized as *middleboxes*, are now software functions that can be migrated to reduce costs, consolidate the processing or scale easily.

But if not carefully implemented, VNFs won’t achieve high-speed and will barely sustain rates of even small networks and therefore fail to fulfil their promise. As of today, out-of-the-box solutions are far from efficient and cannot handle high rates, especially when combined in a single host, as multiple case studies will show in this thesis.

We start by reviewing the current obstacles to high-speed software networking. We leverage current commodity hardware to achieve what seemed impossible to do in software not long ago and made software solutions believed unworthy and untrusted by network operators. Our work paves the way for building a proper software framework for a programmable network infrastructure that can be used to quickly implement network functions. We built FastClick, a faster version of the Click Modular Router, that allows fast packet processing thanks to a careful integration of fast I/O frameworks and a deep study of interactions of their features. FastClick proposes a revised, easier to use execution model that hides multi-queueing and simplifies multithreading using a thread traversal analysis of the configuration. We propose tailored network-specific multi-threaded algorithms that enable parallel high-speed networking. We build a new retro-compatible batching implementation, and avoid system calls “left over” by previous work.

We then build MiddleClick, an NFV dataplane built on top of FastClick. It combines VNFs along a service chain to use a common subsystem that implements shared features such as classification and session handling, but makes sure no feature is applied that isn’t absolutely needed by one of the VNFs. E.g., the classification is optimized to be minimal and only needs to be done once for all VNFs. E.g., if no VNF needs TCP reconstruction, that reconstruction won’t happen. We propose an algorithm to enable a per-session, per-VNF “scratchpad”. Only the minimal amount of state is declared and accessible in predictable locations using a per-VNF offset into the “scratchpad” for fast lookups across the chain.

MiddleClick also offers new flow abstractions and ways to handle sessions that enable fast and easy development of new middlebox functions that can handle many flows in parallel.

Cooperation, consolidation and using the hardware in an appropriate way may not always be enough. This thesis finally explores how to use classification hardware such as smart NICs and SDN switches to accelerate the processing of the combined service chain, removing the need for software classification.

While this work mostly relies on known high-level NFV dataplane principles and proposes a few new ones, it is one of the most low-level work in the field, leading to precise implementation considerations yielding very high performance results. Both FastClick and MiddleClick are available as Open Source projects and constitute an important contribution to the state of the art.

Multiple leading edge use cases are built to show how the prototype can be used to build fast and efficient solutions quickly.

Résumé

Auparavant, les infrastructures réseaux étaient composées de simples commutateurs et routeurs. Mais rapidement sont venu s'ajouter des "middlebox" (littéralement "boite au milieu"). Ces fonctions réseaux (NF pour "Network Functions" en anglais) comprennent les pare-feu, les traducteurs d'adresses ("NAT"), les inspecteurs de paquets, et bien d'autres fonctionnalités destinées à garandir un réseau plus sécurisé, plus performant, ou à rendre le réseau plus intelligent.

Depuis quelques années, ces fonctions réseaux ont progressivement été implémentées au niveau logiciel, en remplacement des appareils dédiés à une seule fonction. Cette tendance est appellée la virtualisation des fonctions réseaux (NFV, "Network Function Virtualization" en anglais). Ces logiciels peuvent être migrer pour réduire les coûts d'exploitation, et facilitent le passage à l'échelle. Les fonctions réseaux virtuelles (VNF) peuvent maintenant tourner dans les centres de données au lieu de matériel spécifique disposé au sein du réseau. Les infrastructures réseaux logicielles promettaient plus de flexibilité, notamment pour utiliser la capacité de calcul potentiellement disponible dans les centres de données ("datacenters").

Mais si les VNFs ne sont pas implémentées précautionneusement, elles ne peuvent pas atteindre la capacité de traitement nécessaire, même pour de petits réseaux. Elles échouent donc à tenir leurs promesses. Aujourd'hui, les solutions disponibles sont peu efficaces et ne peuvent atteindre de grandes capacités de traitement, encore plus quand elles sont combinées dans une seule machine comme le montreront divers cas d'études.

Nous commencerons par passer en revue les obstacles à la haute capacité dans les réseaux logiciels. Nous utiliserons du matériel commun pour arriver à atteindre ce qui semblait pourtant impossible à faire en logiciel il y

a peu et menait à un manque de confiance des opérateurs réseaux dans les solutions purement logicielles. Ce travail mène à la construction d'une "fondation" logicielle pour les infrastructures réseaux programmables qui peut être utilisée pour implémenter tout type de fonctions réseaux.

Dans ce travail, nous développons FastClick, une versions améliorées du "Click Modular Router" (littéralement "Routeur Modulaire Clique") qui permet de faire du traitement de paquet très rapide grâce à une intégration minutieuse de librairies logicielles pour l'entrée/sortie (E/S) et un passage en revue des interactions entre leurs fonctionnalités. FastClick propose un modèle d'exécution revisité et plus facile, car il cache l'utilisation de l'E/S multi-files et réduit les protections nécessaires à la programmation parallèle qui ne sont pas réellement obligatoire en faisant une traversée du graphe de fonctions lors de la configuration du système. Nous proposons des algorithmes de programmation parallèle spécifiques aux réseaux qui permettent un traitement à haute vitesse, un nouveau modèle rétro-compatible de traitement par groupes de paquets, et nous évitons des appels système importants oubliés par la plupart des précédents travaux.

Nous construisons ensuite "MiddleClick", une base logicielle pour la virtualisation de fonctions à partir de FastClick. MiddleClick combine les VNFs d'une chaine de service pour utiliser un sous-système commun qui implémente les fonctionnalités partagées comme la classification et la gestion des sessions, mais s'assure qu'aucune fonction inutilisée ne soie appliquée si elle n'est pas absolument nécessaire pour une des VNFs de la chaine. Par exemple, la classification est optimisée pour être minimale et n'est faite qu'une seule fois pour toutes les VNFs. Si aucune VNF ne nécessite de reconstruction TCP, elle ne sera pas faite. Nous proposons un algorithme qui permet un espace unique par session et par VNF. Seulement l'espace d'état minimal est alloué à des emplacements prédictibles pour que toutes les VNFs aient un accès rapide à l'espace d'état.

MiddleClick introduit aussi une nouvelle abstraction de flux et des moyens de gérer les sessions qui permettent un développement facile et rapide de nouvelles "middlebox" qui peuvent traiter de nombreux flux en parallèle.

La coopération et la consolidation de fonctions de même que la gestion efficace du matériel ne sont pas toujours suffisantes. Cette thèse explore des méthodes pour utiliser du matériel en amont comme un commutateur “SDN” ou une carte réseau intelligente pour accélérer le traitement et les chaines combinées, en enlevant le besoin pour la classification en software.

Ce travail se repose sur des principes de virtualisation de fonctions réseaux et en propose de nouveaux. Mais c'est surtout en proposant un passage en revue plus bas niveau que ce travail se détache. En découle une implémentation avec des considérations très spécifiques aux réseaux à hautes capacités menant à de très bonnes performances. FastClick et MiddleClick sont tous deux disponibles en tant que logiciels libres, ce qui constitue en soit un apport à l'état de l'art.

Plusieurs cas d'études novateurs sont étudiés pour montrer comment le prototype peut être utilisé pour construire des fonctions réseaux efficaces, et ce très rapidement.

Acknowledgements

Getting to this thesis has been a long journey, full of collaborations, discussions, and many hours, days and months of testing. I learned from many people from the USA to China, through Sweden. They helped me to get to the final results presented in this thesis.

I'd like to particularly thank my thesis advisor, Professor Laurent Mathy who helped me with wise guidance on technical solutions to overcome many challenges, and my colleague Cyril Soldani who took part in the elaboration of previous and undergoing work, many pieces of which are presented here. Of course, other colleagues have provided many interesting meetings and lunch-time discussions allowing me to import numerous bits of other fields to my work.

In the last year, I started trying some of my work in Cisco Meraki security appliances, leading to various interesting findings, in this real-case application. But the biggest contributions from that exchange are the en-lighting talks, so much so that I stayed in touch just to hear their scaling problems and to have great fun trying to solve them.

I would like also to thank the University of Liege and its IT service (SEGI) for authorizing traffic trace captures of network data (under strict terms) to prove that my findings do not apply only to spheric chickens in a vacuum.

This work has been funded by the Belgian *Fond National de la Recherche Scientifique (FNRS).* This funding gave me the rare opportunity to build a fully-fledged lab as my needs grew. In the last few months, I was also part of the Superfluidity European H2020 project.

This work is built on multiple open source software cited through the manuscript. I would like to thanks all the contributors and people of their community for helping me here and there, mainly the Click Modular Router community.

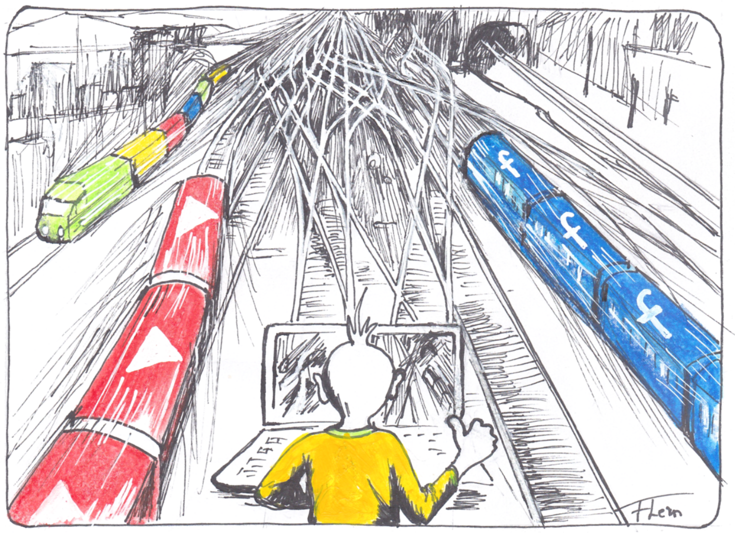
Finally, last but not least, I would like to thank my family and soulmate for their support and the provided food during the busiest times. My mother deserves a special mention for providing the illustrations that enhance this thesis.

Contents

List of Figures

List of Tables

Programmable Network Infrastructure



1 Introduction

At the beginning, networks were essentially composed of end hosts - *computers and servers* - and forwarding machines, - *switches and routers*. Hosts communicated with each other, and machines in the middle were only concerned with the data - *the packets*. Except for small changes for the sake of correct routing[1], the traffic between the hosts was left untouched. In a sense, the network infrastructure was composed of the minimal functionality, leaving most of the duty to the machines at the edge. This is referred as the end-to-end principle as depicted in figure 1.1.

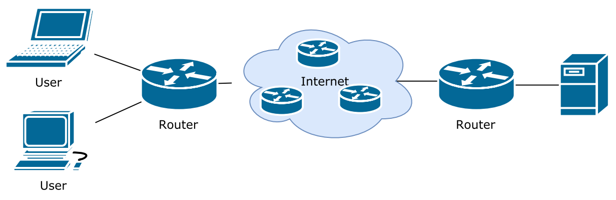


Figure 1.1: Untouched end-to-end network, with only switches and routers between hosts that receive data exactly as it was sent

But quickly, for security but also for many different needs, other machines were introduced in the middle of the network. For instance, some of them block some unallowed types of packets, they are *firewalls*. Others analyze the packet content - *deep packet inspectors (DPI)* - to decide if they are attacks or banned content that should be rejected before it reaches the final recipient. All those devices that act “in the middle” are referred to as *middleboxes*[1]. Example of such network, more representative of

[1]For instance, the Time To Live (TTL) field is modified by routers to protect against endless loops

today's state of the Internet, is depicted in figure 1.2. End-users often have a *Customer Premise Equipment (CPE)* in their homes that are provided by the *Internet Service Provider (ISP)* and run multiple functions typical of middleboxes such as a firewall or a *Network Address Translator (NAT)*, that allows hiding internal devices behind the CPE as if it was the only equipment connected to the internet. The ISP network may include another round of security device, *Wide Area Network (WAN)* optimizers, especially mobile providers, that breaks the end-to-end principle on purpose to cope with the different rates of cable networks and wireless networks. Most computers that only provides Internet services, called servers, are grouped in farms called datacenters. Datacenters generally run stateful firewalls that retain information about specific hosts to limit the number of connections to prevent *Denial of Service (DoS)* attacks. Datacenters often use *Load Balancers (LB)* that take care of dispatching traffic to multiple servers to spread the load. In which case in the end-to-end principle, the second end may actually change between multiple connections without the other end knowing it.

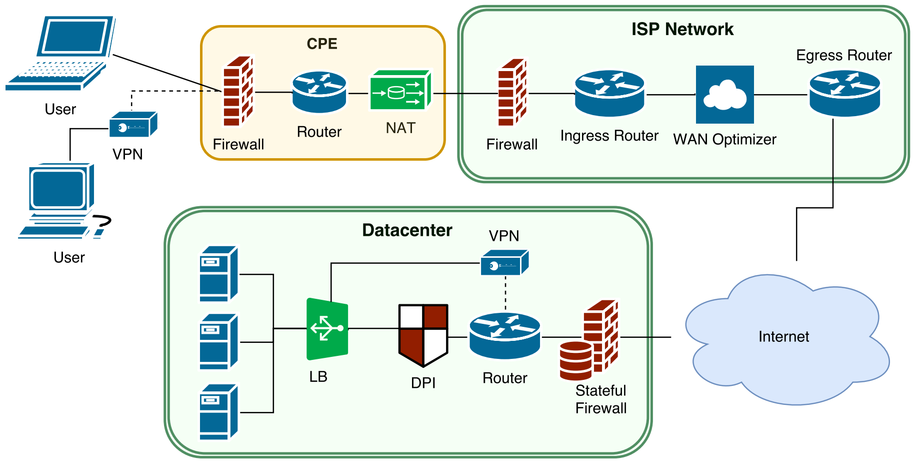


Figure 1.2: A more realistic network, comprising a number of middleboxes that heavily modify the traffic

This is only a limited non-exhaustive list of middleboxes that can be found in the network, [1] categorized 22 type of middleboxes in 2002, a number which would probably be higher today. According to [2, 3] middleboxes account now for 1/3 of the network devices in enterprises network and are also massively present in datacenters.

Middleboxes, however introduce multiple problems.

Network ossification. Middleboxes modify the traffic, breaking the end-to-end principle [4]. Some of them drop unknown type of packets or do not handle correctly new protocols, preventing innovation in the network [5].

Laborious development There is currently no good, widely deployed middlebox programming framework that allows fast development of stateful network functions. Multiple propositions in the state of the art will be reviewed in this work, but developing a middlebox is not an easy task as networking stacks do not handle easily high level of parallelism (OSes performance are reviewed in chapter 2). They do not allow flexible flow definition (in general only TCP and UDP) and do not handle session management, that is allowing the programmer to remember data per-session. Indeed, middleboxes often need to keep specific statistics or state for each different micro-flows. That leaves developers handling multi-thread contexts, hash-tables for session data, and makes it difficult to achieve high-speed when a high number of flows is involved.

No cooperation between middleboxes Most middleboxes exchange packets as RAW packets, without any meta-data and thus the same work is repeatedly done by various middleboxes even inside the same operator-owned network.

They are responsible for a lot of failures In a large scale review[3], it was shown that "Middleboxes contribute to 43% of high severity incidents despite being 11% of the population". Some functions make failover harder or nearly impossible, especially stateful functions. For instance, NAT failover would require the flow table to be shared between multiple appliances to allow mapping of micro-flows to keep the same translation when the second one takes over.

High-speed needs hardware. To handle large amount of traffic, most high-end middleboxes are hardware-based. Major innovation needs a change of hardware, as seen with the IPv6 deployment which required most hardware equipment to be changed. Moreover, hardware equipment is often costly, not necessarily supporting fault tolerance. And even when they do, the spare equipment does not always take over when a fault occurs[2].

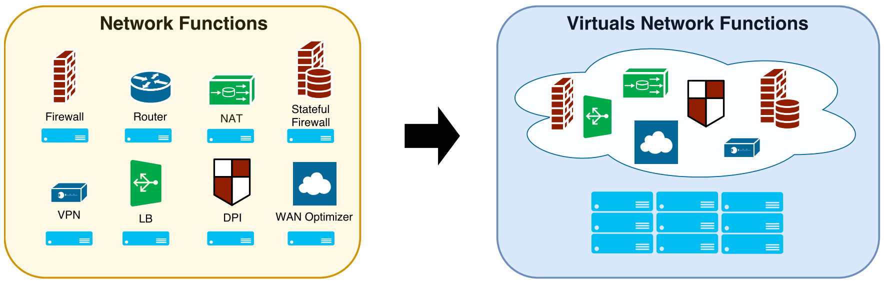


Figure 1.3: From physical boxes implementing network functions to virtual network functions (VNFs)

1.1 Network Function Virtualization

Therefore as networks are composed of more and more complex equipment performing legitimate and needed work, but given that they ossify the network and they are responsible for lots of failures, the last few years have witnessed the migration of hardware implementations to much more flexible software counterparts, a trend called *Network Function Virtualization (NFV)* shown in figure 1.3. NFV enables faster innovation[6], more programmable network infrastructure and allows to take advantage of the high computing power available in the cloud.

Software packet switching and routing started the NFV trend. With early research about high-speed packet header classification algorithms[7, 8] that would perform well in software. But the best classification algorithms would still achieve a much lower packet processing rate than their hardware counterpart[9, 10]. Software routers reached a momentum with the Click Modular Router[11]. Even if the speed of Click was not comparable to its contemporary hardware routers, the modularity it provided made it popular, being still of today one of the most used packet processing prototyping platform for researchers[12, 13]. Click did not only exceeded performance of its contemporary Operating System's IP routing stack[14], but offered an easy-to-use, graph-based programmable packet processing platform that lend itself for general purpose network function virtualization. However, even if Click achieved sufficient performance to replace low-end and mid-end routers[11, 14], it was still not fast enough to replace high-end routers that can be found in the core networks or datacenters.

The challenge is therefore to enable high-speed software packet processing, while keeping the advantage of software flexibility.

No matter how much optimization comes into place, software scalability requires multi-processing at some stage. The network functions can be split over multiple cores but also even across multiple servers. Subsequent work[15, 16, 17, 18] therefore studied parallelisation of software packet processing platforms and fairness while consolidating virtual routers on a single system[19, 20].

In this thesis we also present an extension of Click, called FastClick, that provides an automatic parallelisation model. FastClick also leverages latest commodity hardware features to enable very high-speed packet processing after conducting a very extensive review of their interaction. The techniques behind FastClick, such as hardware-specifics for fast I/O, possible scaling and execution models, how to efficiently handle packets with batching and zero-copy, will be further discussed in chapter 3 and form an important contribution of this thesis. FastClick is available in Open Source at [21], and provides by itself a piece of software useful to the research community as a basis for further development[22, 23, 24, 25] or a relevant point of comparison as an established state of the art packet processing platform[26, 27, 28, 29].

In chapter 4, we review different models to distribute packets among multiple cores, extending the analysis of [16] and [30] with various other considerations about the impact of different pipelining models, amount of memory accesses and their location and number of CPU cores. We also study and propose specific data-structures to protect mutable states in the context of high-speed networking, how to detect concurrent accesses, and how to ensure that unsafe parts of the graph are not traversed by multiple threads. We find the best spot in use cases for each data structure and provide an openly available implementation at [21].

Most MiddleBoxes functions mostly supervise the network traffic and do not initiate or close connections. However, usual Operating Systems network stacks are implemented in the scope of packets being received and consumed by a socket application that terminates the connection. Therefore a lot of heavy work is done upfront, directly in the driver, which is not necessary for “pass-through” workload. Multiple userlevel I/O frameworks such as DPDK[31], Netmap[32] and other initatives[33, 34, 35, 36] were proposed to deliver network packets to userlevel, bypassing the Kernel to achieve high-speed. Chapter 2 conduct an extensive review of those frameworks, how they work, and the techniques used such as interrupt mitigation and batching of packets to achieve high packet reception and transmission rate. It is our belief that our review will help the reader understanding the ups and downs of each framework. Our contribution also includes a quantitative comparison of those frameworks. Section 2.4 explores how to re-mediate to the lack of fast path in Operating System network stacks, and proposes improvements to the Linux Kernel to enable in-kernel high-speed packet processing. Section 3.4 conduct a review of non-networking I/O system calls to avoid when building network functions in userlevel such as time reading system calls. We build a new userlevel clock that allows reading the time much faster than when using kernel facilities, with a very high precision. While the precited recent I/O frameworks focus on network I/O and similar techniques have been used for management of storage devices[37], we propose a userlevel clock that synchronizes upon the operating system time as part of FastClick at [21].

It is our belief that the platform we built has revisited those many aspects of high-speed packet processing and is a strong basis to build new scalable virtual network functions.

1.2 Stateful service chaining

In general, multiple network functions are chained together to form a *service chain*. Running service chains with pure NFV allows operators to move the functions to run them in different equipment or datacenters. A possibility helped and pushed by the recent trend called *Software Defined Networking (SDN)* that allows to program switches using software controllers, and direct each flow specifically through the network. With SDN, each flow can easily follow a different service chain.

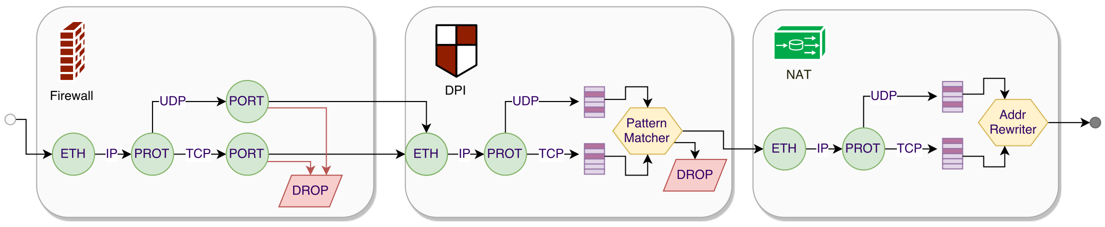


Figure 1.4: A standard service chain with firewall, DPI and NAT decoupled in basic blocks. One can easily see the case for factorization.

Figure 1.4 represents a simplified logical view of a service chain composed of a stateless firewall, a DPI and a NAT. They are decoupled in basic blocks. They all

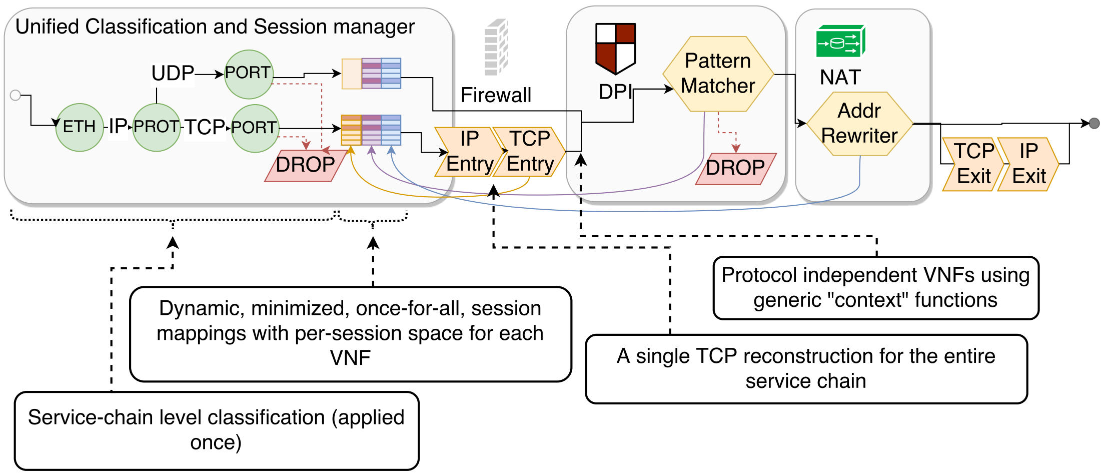

Unified Classification and Session manager
UDP
PORT
ETH
IP
PROT
TCP
PORT
DROP
Firewall
IP Entry
TCP Entry
DPI
Pattern Matcher
DROP
NAT
Addr Rewriter
TCP Exit
IP Exit
Protocol independent VNFs using generic "context" functions
Dynamic, minimized, once-for-all, session mappings with per-session space for each VNF
A single TCP reconstruction for the entire service chain
Service-chain level classification (applied once)

Figure 1.5: A improved version of the service chain of figure 1.4

start with some traffic class classification steps that dispatch packets according to their headers. In fact, a stateless firewall is only about classification. The DPI will also do some classification to run some protocol-based checks and then use a session table to rebuild micro-flows and look for patterns inside the reconstructed payload. The NAT will use the same kind of session tables to remember the mapping to apply for each session. All classification steps in those middleboxes are mostly similar, while the session tables are almost identical.

The *first challenge* to enable efficient service chaining is therefore to propose an infrastructure which can factorize these redundant pieces of work, to avoid a fall in performance as the service chain grows.

xOMB[38] decouples network functions to allow better programmability, SNF[39] to combine some identical basic read and write operations while CoMb[40] explores high-level, per-block consolidation of middleboxes for better resources management. We extend those ideas by proposing a programmable infrastructure that allows VNFs to expose their classification and per-session state needs. The system can then factorize the classification and reconcile the state for all middleboxes as shown in figure 1.5. The unified state management avoids inconsistent state and prevents to have multiple session tables that will essentially be the same along the service chain. Moreover, the work to be done by the platform for each VNF is minimized by the process of factorization and allocate per-session space that is needed for each VNF at once. Only the minimal amount of state is declared and VNFs are informed of a constant offset to look at in the unified session space for fast lookup. The unified session management and the session classification combination for all VNFs of the chain is one of our major improvements over the state of the art which mostly combine the traffic class, static classification[39, 41]. The unique session table leads to very high-speed throughput, even for very long and diversified service chain and offer potential acceleration for stateless functions such as firewalls that may use the session table that is present anyway to store per-flow decision.

The *second challenge* is to avoid multiplying identical protocol-specific stack services along the chain, such as reconciling TCP state, reordering TCP packets or joining and splitting a stream of bytes into packets.

NetBricks[42] and E2[43] allow to chain functions, passing TCP streams instead of packets between the components, avoiding re-construction of TCP sessions. In the system we present, we propose a generilized (*i.e.* not TCP-centric) version of this idea. Keeping the protocol stream is automatized and not the result of specific pipelining of VNFs built using a specific stream APIs. All VNFs exchange batches of packets of the same session, but without losing the associated session context. The user may then use various layer of abstractions described further in section 5.5 that allows to work on packets as a stream of bytes. The VNF are all compatibles as they always exchange packets underneath the bytestream abstraction. To avoid multiple identical stack services, these abstractions makes request to a given current protocol "context" that is set using specific building blocks that are easily interchangeable. *E.g.* one can build a pattern-matching VNF in only a few lines using the higher level abstraction that provides an iterator over the payload of bytes, reminescent of FlowOS[44]. Therefore, multiple VNFs working on the same context can be chained without any protocol-related performance cost.

The *third challenge* regarding stateful service chaining is to allow for innovation, and be, as much as possible, future-proof. Most high-speed NFV dataplane supporting stream modification come with a userlevel TCP stack[27, 42, 45, 46], that aside from implementing only the TCP protocol, are hard to maintain, or already strip some options of TCP preventing innovation and future deployments. Some TCP stacks implement older or fixed congestion or flow control algorithms that will not work well with future ones.

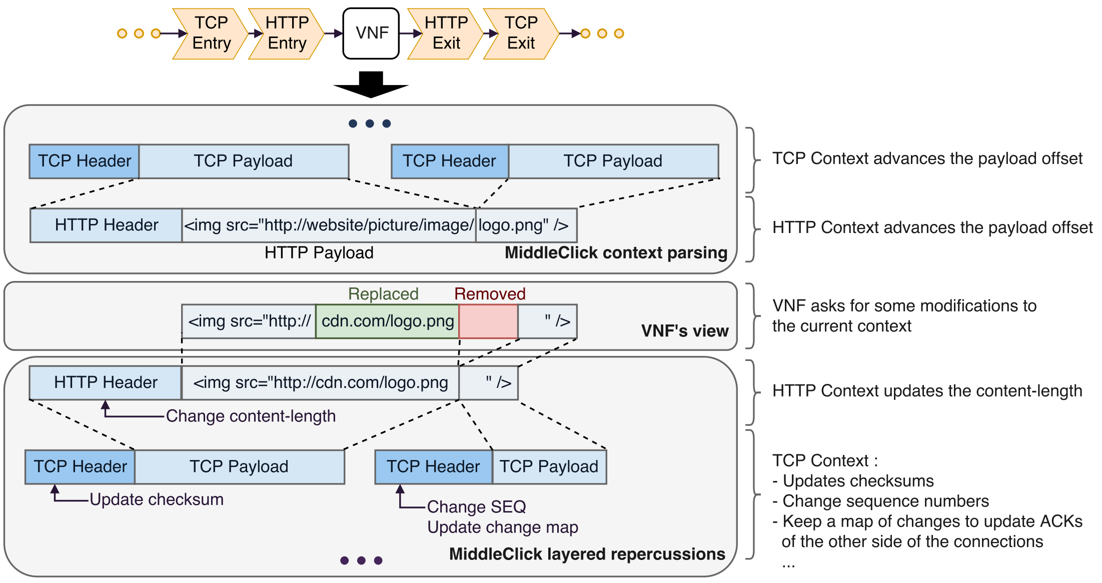

TCP Entry
HTTP Entry
VNF
HTTP Exit
TCP Exit
TCP Header
TCP Payload
TCP Header
TCP Payload
HTTP Header
<img src="http://website/picture/image/
logo.png" />
HTTP Payload
MiddleClick context parsing
TCP Context advances the payload offset
HTTP Context advances the payload offset
Replaced
Removed
<img src="http://
cdn.com/logo.png
" />
VNF's view
VNF asks for some modifications to the current context
HTTP Header
<img src="http://cdn.com/logo.png
" />
Change content-length
HTTP Context updates the content-length
TCP Header
TCP Payload
TCP Header
TCP Payload
Update checksum
Change SEQ
Update change map
MiddleClick layered repercussions
TCP Context :
- Updates checksums
- Change sequence numbers
- Keep a map of changes to update ACKs of the other side of the connections
...

Figure 1.6: MiddleClick flow abstraction system. A VNF can make requests to its current abstract context that takes care of the implications for the protocol it supports, and then pass the request to the lower context and so on. This allows to easily build support for tempering flows of new protocols on top of others.

The context-based abstraction provides seamless inspection and modification of the content of any flows (such as TCP or HTTP), automatically reflecting a consistent view, across layers, of flows modified on-the-fly as shown in figure 1.6. When an HTTP payload is modified, the content-length must be corrected. A layered approach allows to back-propagate the effect of stream modification across lower layers as shown in figure 1.5. The stack can modify on-the-fly sequence and acknowledgement numbers on both sides of a TCP stream when the upper layer makes changes. The advantages of this system are in two parts. Firstly, we do not need to implement a full TCP stack, as we rely on the end hosts to implement congestion, flow control and handle most retransmissions, therefore protocol blocks are very thin, and easier to maintain. We avoid the overhead of a full TCP stack and do not make assumptions about the TCP implementations at the ends. Secondly, each block layer is easily interchangeable, and can be added on top of another layer to support new protocols, making it easy to implement new application layers on top of HTTP, or new flow control on top of UDP such as UDT[47]. Aside from VNFs tightly tied to the underlying protocol (*e.g.* WAN optimizers), no change in the VNFs themselves needs to be done when changing the context blocks in front of them.

The system finally provides support for a mechanism to “wait for more data” when a middlebox needs to buffer packets, unable to make a decision while data is still missing. Our TCP in-the-middle implementation supports pro-active ACKing to avoid stalling a flow while waiting for more data, and enables handling of large amounts of flow using a run to completion-or-buffer model. The ability to easily save per-session states without the cost of a context switch allows to build innovative function such as DPIs that are not subject to eviction by saving the state of the DFA in the session table.

Chapter 5 focus on building such a factorization and abstraction framework that is shown to be efficient and enables fast development of complex flow-based functions in no-time. While the examples VNFs built using the platform perform better than the compared state of the art, a big contribution of this work to the field of high-speed packet networking is also the availability of MiddleClick, our prototype that builds upon the efficiency of FastClick which can be used to quickly implement stateful network functions. It is fully available at [48] with the hope it will be used by others to build new efficient VNFs.

As the needs of the Internet grows, service chains and the complexity of the VNFs themselves. Therefore multiple researchers tried to offload some part of the VNFs processing to GPU[13, 35] or FPGA[49].

In this work, we try to offload the classification needed by the VNFs to hardware facilities. Indeed, the factorization of the classification needs by MiddleClick offers a unique opportunity to move it even before the traffic hits the CPU. Chapter 6 study how various hardware can be used to accelerate MiddleClick by offloading some part of the classification. We also present a way to not only offload the service chain classification but let the hardware tag the packets so it can be received by the right CPU core that handles the service chain directly, avoiding any useless inter-core communication in section 6.2. This contribution therefore leaves those CPU cycles free to do the meaningful processing of the VNFs, achieving 100Gbps processing[25].

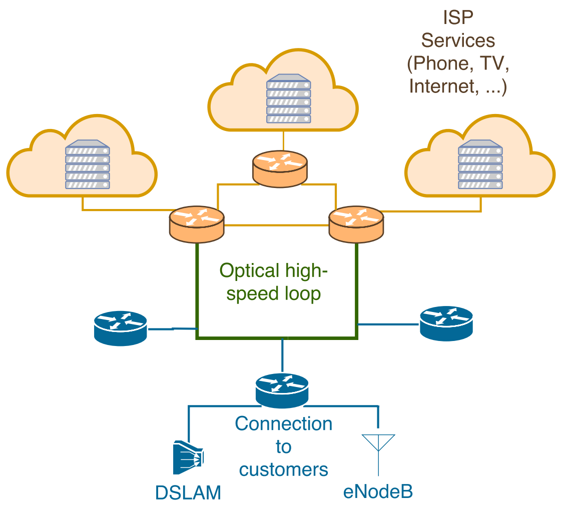


Figure 1.7: Simplified view of the TeraStream network where most services of the Deutsch Telekom ISP are moved to datacenters

1.3 Use cases for a high-speed programmable infrastructure

In their TeraStream network[50], Deutsche Telekom uses an inter-connected IPv6 only network to connect all customer access interfaces (LTE, DSLAM, ...). This is a pure optical switching network, connected to datacenters to run the "Infrastructure Cloud". This is represented in a simplified way in figure 1.7. In that way, they remove all services from the networking equipment and are able to fully scale them, instead of relying on "intelligent routers" (routers embedding middlebox functions) to do the processing along the path. Virtual middleboxes avoid the need to have more expensive network equipment than switching gears in the core of the network as Deutsche Telekom does.

Virtualizing the infrastructure by moving middleboxes themselves to NFV enables middlebox consolidation in datacenters to reduce energy consumption. A large *Content Delivery Network (CDN)* provider, Akamai is moving computation to datacenters where the energy price is currently cheaper[51]. Nowadays the bandwidth in the core network is not the main problem, simple high-rate switches and routers that only do switching can transfer the job to be done to network edges or other datacenters efficiently and at affordable costs because of their relative simplicity as they only do L2 or L3 switching.

Energy consumption is, however, a challenge to solve. Therefore CDN providers can afford to pay ISPs for bandwidth and move requests around continents.

An infrastructure easier to program and more flexible ease the development of the two examples above. But it also enables novel use cases such as carrier-grade ad-removal or user-targeted ad-insertion, on the fly video re-encoding to cope with bandwidth changes, parental filtering that is based on the content of the flow and not only headers, better innovative DDOS protection, application-level proxies and caches and many others that only need to be invented. One could modify a flow to provide information for augmented reality, showing the user targeted information on his screen. The proposed framework also builds a ground for a super-fluid network where VNFs can move from one place to another. In such network, functions usually executed on a mobile such as ad-removal, parental control or firewall can be moved to the network edge to improve security and battery.

While there is currently no standard for 5G deployments, one way to achieve the very high rate is to allow mobile devices to communicate with multiple antennas using a MIMO technology. That means the data must be re-conciliated in the *Radio Access Network (RAN)* by middleboxes. To accommodate speeds as high as 10Gbits per mobile devices, most middleboxes functions must be pushed to the network edge as the WAN bandwidth will not be able to achieve the aggregated rate to transport all traffic to datacenters. Therefore the 5G network makes a big case for fluid software networking where the proxy caches, proactive content caching, WAN optimizers or even third party VNFs can run in the operator's owned devices between the RAN and the *Core Network (CN)*.

NFV also allows redundancy by taking advantage of virtualization technologies largely available in the cloud, allowing easy scalability as the VNF can spread over multiple CPUs or even multiple compute nodes. NFV also enables the concept of *Infrastructure as a Service (IaaS)*, where instead of running its own middleboxes a network operator may delegate middlebox functions to a third party, cloud-based provider as proposed in [2], who estimated that 90% of network functions present in enterprises networks could be run by third parties in the cloud. In this work, we build a high-speed packet processing platform, FastClick, that we then enhance to allow building innovative and efficient stateful service chains. We tackle in this work all the observed MiddleBox challenges highlighted at the beginning of the chapter. We therefore believe

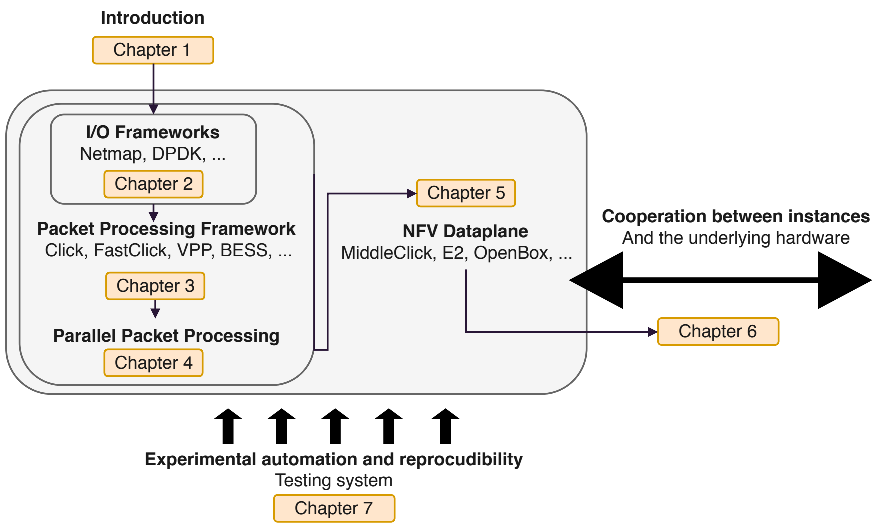

Introduction
Chapter 1
I/O Frameworks
Netmap, DPDK, ...
Chapter 2
Packet Processing Framework
Click, FastClick, VPP, BESS, ...
Chapter 3
Parallel Packet Processing
Chapter 4
Chapter 5
NFV Dataplane
MiddleClick, E2, OpenBox, ...
Cooperation between instances
And the underlying hardware
Chapter 6
Experimental automation and reprocudibility
Testing system
Chapter 7

Figure 1.8: This thesis follows a bottom-up approach to explain how we build our architecture for programmable infrastructure

the solution presented in this thesis is fit for purpose and will contribute to the state of the art towards, a fluid, programmable, high-speed network infrastructure.

1.4 Structure of this thesis

Chapter 2 to 6 approaches the problem of building a high-speed programmable infrastructure with a layered, bottom-up approach instead of a more tradition unique state-of-the-art ⇒ solution ⇒ evaluation approach as shown in figure 1.8. Chapter 2 reviews the state of the art of **pure network I/O** and compare the available frameworks for fast high-speed packet transmission. Chapter 3 studies the ways to integrate those methods in a **high-speed I/O platform** that allows high-speed packet-based processing and then how to scale using more CPU cores in chapter 4. Chapter 5 builds on top of that platform to provide proper NFV features such as flow processing and compares to other **NFV architectures** proposed in the state of the art. Chapter 6 reviews how to use multiple, maybe different, hardware environment to enable **cooperation** of multiple hosts, expanding the programmable infrastructure over the whole network of an operator. Therefore it is worth mentioning that given that layered approach, detailed comparison with the state of the art and highlighting of our specific contributions are also categorized by chapter. We have taken the problem layer per layer, reviewed the state of the art, implemented solutions on top of it to make it faster, evaluated what we've done and moved to the next step. *I.e.* high-speed I/O is needed to build a fast network platform, which should be fast enough to build an NFV platform, and only when good enough will make a case for making multiple of them cooperate.

Chapter 7 details how experiments were done. For the purpose of our research, most of them were done using a hand-crafted tool that is also described as part of this thesis. This tool also contributes to the field of high-speed networking by allowing to write performances tests in an easy language. The tool allows studying the behaviour of solutions under development in term of latency, throughput and other metrics. It is parametrisable to review how the metrics change under various parameters. The tool processes the huge amount of samples that a big number of parameters leads to allowing a human to quickly understand how the device under test reacts to parameters change.

Chapter 8 finally discusses further possibilities and concludes about our work.

This chapter in a nutshell

▶ **Context of this thesis**

- NFV is promising, especially to solve network ossification problems. NFV helps to leverage the power available in the cloud and prepare for the network of tomorrow such as 5G mobility.
- But current implementations are very much perfectible in term of performance, ease of programming and ability to build efficient service chain.

▶ **Highlight of our main contributions.**

- We review high-speed I/O state of the art and compare available frameworks.
- We leverage known techniques to build a high-speed platform, make a novel study of their impact and interactions deeper than the state of the art.
- We add new techniques of our own to achieve high speed, in a safe but scalable multi-threaded context leading to FastClick, now used in multiple recent publications as a faster version of the Click Modular Router.
- We build an NFV dataplane that is very efficient and performs better than current approaches using a novel classification system and facilities that allows

both easier development and factorization of common functions in a unified, faster and hardware offloadable flow manager. The dataplane is automatically tailored to the needs of the VNFs that compose the service chain.

- The platform allows to modify TCP (but also others) flows on the fly without the need to terminate the connection, leading to better performances than proposed in the state of the art.
- We prove the case for middlebox cooperation to build a programmable network infrastructure.
- We build an experiment automation tool suited for high-speed networking (NPF).
- Our main open source contributions, FastClick, MiddleClick and NPF are available online at [21], [48] and [52].

2

The fall of the old paradigms

In this section, we review the current industry practices, and the model followed by Operating Systems to receive and transmit packets, how to improve this model and the alternatives.

Standard Operating System kernels such as Linux are built with functionality in mind. They offer full-stack services and are shaped with the scope of packets ending up in a socket, being delivered to a userspace application. While this suits well end hosts, it may not be the best approach in an NFV context where most of the VNFs do not need to terminate connections. Most VNFs only analyse the data passing through, or slightly modify it and therefore do not need to fully terminate a connection and re-open a new one. Section 2.1 reviews usual operating system network stack implementations and their bottlenecks.

In recent years, we have witnessed the emergence of high speed packet I/O frameworks such as Netmap[32] or DPDK[31], bringing unprecedented network performances to userspace. Section 2.2 reviews a set of existing userspace I/O frameworks and their mechanisms to achieve better throughput and latency. Section 2.3 then evaluates their forwarding performance.

Most solutions proposed by those frameworks are possible inside the kernel itself. But even if performance can be improved, building Kernel application is complex and the slightest mistake can crash the kernel. Nevertheless, some software like the Click Modular Router[11] can run in-kernel. In section 2.4, we propose a set of changes and build upon novel features of the Linux Kernel to make it suit better high-speed networking applications. We build a proof-of-concept to evaluate potential performance gains that shows a 4X improvement with minimal size packets. The changes made also prevent the livelock problem that will be described in section 2.2.1 that completely stalls the system when under high receive rate.

This chapter in a nutshell

▶ **Context of this chapter**

- Operating Systems are generally made for terminating connections, they are not suitable as an infrastructure platform that barely modifies packets and passes them as fast as possible.
- A lot of new frameworks allow receiving packets directly in userspace very quickly, by-passing the OS network stack and allowing easier programming than in-kernel.

▶ **Highlight of our main contributions in this chapter**

- We review the techniques for high-speed I/O and the frameworks using those techniques.
- We evaluate the different methods to achieve high-speed packet processing and compare their performances.
- We propose some slight but conceptually important modifications to the Linux Kernel to allow fast in-kernel processing.

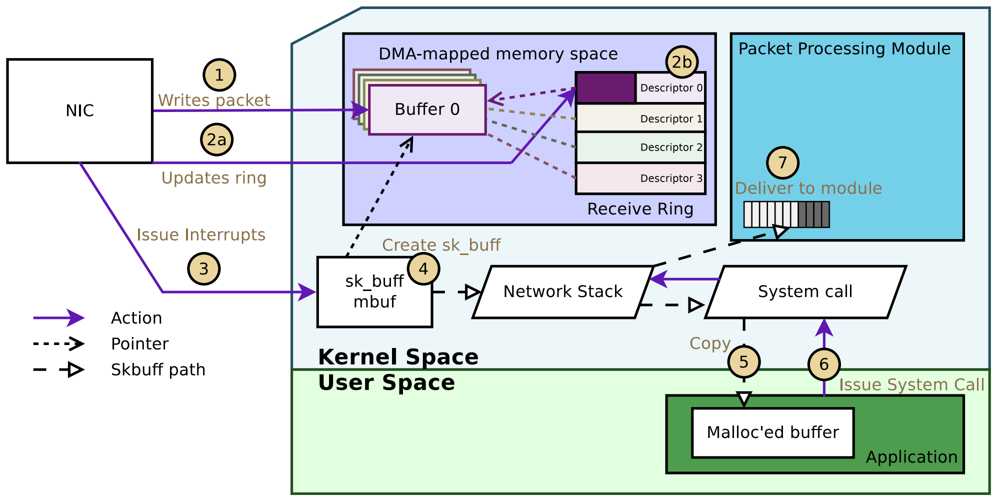


Figure 2.1: Usual network I/O receive path for userlevel or kernel applications

2.1 Kernel I/O

This section will briefly summarize the packet receive path of common Operating Systems. Of course, exact details depend on the specific OS, and even inside a given OS a fair part of the path actually depends on the driver.

The driver generally arranges for a *Direct Memory Access (DMA)* mapped memory space visible to both kernel and the *Network Interface Controller (NIC)*. The DMA memory space contains some buffers that will receive the packet content, and rings of packet descriptors. The descriptors reference those buffers and information about the data in the buffer, such as the packet length and various kind of flags(fig. 2.1 - 2b).

When a packet is received, the NIC directly writes the packet inside a buffer of the DMA memory space (fig. 2.1 - 1) using DMA. When the copy is finished, the NIC updates the receive ring (fig. 2.1 - 2) to set the length of the packet it just copied, along with a few other information. The NIC will then issue an *Interrupt Request (IRQ)* that will make the CPU stop its current task if interrupts are not masked and run the interrupt request handler of the NIC driver(fig. 2.1 - 3).

The NIC may issue an interrupt request after each packet, after a batch of packets has been written, or none for some time if technologies like interrupt throttling or *New network API (NAPI)* in the Linux Kernel are used to lower the interrupt rate. When NAPI is used, the IRQ handler will disable further IRQ. A kernel thread will then poll for incoming packets, reading the DMA memory zone to check if packets were received. After a certain amount of packets or if no more are available, it will unschedule and re-enable interruptions.

The driver's packet receive loop (executed under NAPI or the IRQ handler) and other subsequent routines build an internal Kernel packet descriptor around the buffer. On freebsd it is the mbuf structure and on Linux it is the sk_buff structure. We will keep sk_buff to reference this kind of structure for convenience, as the idea is the same for most OSes. Some driver pre-create sk_buff around the buffer, so the structure is nearly ready to go when the interrupt happens. The driver then pushes the sk_buff through the network stack(fig. 2.1 - 4).

In the end, a list of "pending" sk_buff is built in kernel memory, either for delivery to third-party kernel application modules(fig. 2.1 - 7) or to userspace.

In the case of userspace networking, the application issues read/write system calls, passing userspace buffers(fig. 2.1 - 6). The system call handler will copy the data from the sk_buff to the userspace application's buffers(fig. 2.1 - 5).

[53] analysed in 2006 the receive path of the Linux kernel and is still a valuable source for further reading as most of the process is still identical.

2.2 Kernel by-pass networking

Contribution notice

Most of section 2.2 is from published work [54] made in collaboration with Cyril Soldani and Laurent Mathy.

One man cannot solve all the world's problems ■

Recent years have seen a renewed interest in software packet 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 NICs, such as support for multi-queueing.

We first 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 kernelspace 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, where the data is copied 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, come at a performance cost which prevents to sustain linerate 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[55].

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-queues 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.

Lighter interrupt, or no interrupt. Under very high throughput, it may happen that the CPU runs interrupt routines to handle packets without having any time left for the application to run and actually consume the packets. This is referred to as receive livelock. It is often mistakenly thought that IRQ throttling and NAPI can solve this problem. While they avoid IRQ storms when the rate is high, the networking routine that fills the sk_buffs will still run as a kernel thread (as a softirq in Linux or "bottom halves"), copying packets or pointers to some userspace buffers if a socket is open, with no way to apply *backpressure* to the NIC when those buffers are full.

The result is that the driver receive handler will still receive packets and initialize sk_buffs that cannot be processed by the application endlessly and end up being dropped. [56] already showed this problem and that beyond a certain packet rate the performance drops. While NAPI is already a response from the Linux Kernel to receive livelock, it actually fails to solve it. [57] tuned the process scheduler, the process priority and the NAPI packet budget to lower the livelock effect for the Snort IDS running in userlevel. We ran an experiment similar to theirs in figure 2.2. We use 4 cores, each of them reading packets from one of the 4 10G NICs (Intel 82599 chipsets) in parallel, transmitting directly the packets using PCAP. The PCAP application only rewrites the MAC addresses of the packets and transmits the packets as soon as possible. Our findings are different than theirs as the priority is more relevant than the budget in our case. We explain this by the fact that the good tuning depends on the driver (their hardware used the e1000 driver while our is ixgbe), on the input traffic characteristic,

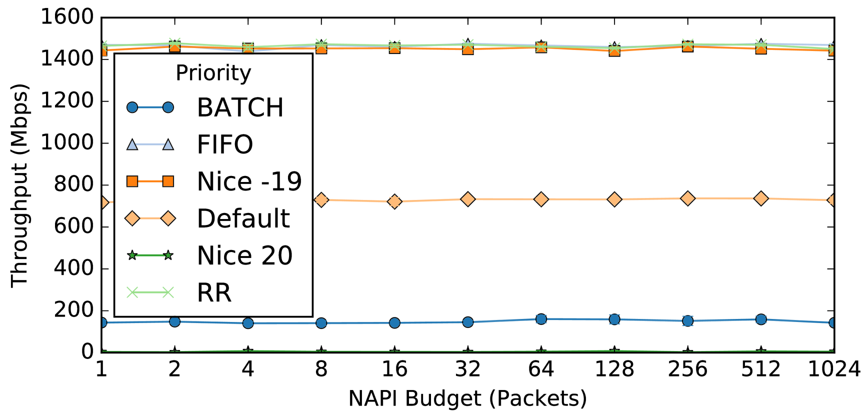


Figure 2.2: Trying various userlevel process priorities/schedulers and NAPI budget for a forwarding application where each of the 4 10G NICs interrupts are pinned to the same 4 cores used to run the application. The application simply forwards packets to one of the NICs, potentially achieving 40G of 64bytes UDP packets. Further testbed description is available in section 2.3, with the exception that the Linux version used for this test is 4.9 and the CPU is a Xeon E5-2630 v3 @ 2.40GHz

the packet rate but also the size of the packets and the time taken to process each packets. Moreover lowering the NAPI packet budget augments the time the CPU spends in masking and unmasking interrupts and rescheduling the NAPI task, leading to less useful CPU work. Making the application run inside the kernel itself (as in figure 2.1 - 5) will lead to the same priority problem.

Some frameworks therefore drastically reduce or completely remove the interrupt work. This allows userspace applications to run even under a high packet rate. The applications then call the framework routines which will do the job that the interrupts had to do usually in the Kernel stack. As the application themselves will not call the receive routine if they don't have enough time to process the packets, the method effectively applies *backpressure*. That is, when the packets are not served, the NIC stops receiving packets as the NIC ring becomes full.

2.2.2 I/O Frameworks

We first review the technical aspects of some existing userspace I/O frameworks, such as Netmap[32], the DPDK[31], OpenOnload[34], PF_RING[33], PacketShader I/O[35] and Packet_MMAP[58]. Some other works go further than a "simple" module bypassing the kernel, like IX[59] and Arrakis[37]. We won't consider those two in this section as, for our purpose, they only offer fast access to raw packets, but in a more protected way than the other I/O frameworks, using virtualization techniques. They will be reviewed as potential other NFV platforms in chapter 5.

Framework	Packet_mmap	PacketShader I/O	Netmap	PF_RING ZC	DPDK	OpenOnload
Zero-copy	~	N	Y	Y	Y	Y
Buffer pool unique for RX and TX	N	Y	Y	Y	Y	Y
Kernel bypass	N	Y	Y	Y	Y	Y
I/O Batching	Y	Y	Y	Y	Y	Y
Hardware multi-queues support	N	Y	Y	Y	Y	Y
Devices family supported	ALL	1	10 ZC / ALL (non-ZC)	4 ZC / ALL (non-ZC)	32	All 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 2.1: I/O Frameworks features summary.

Packet_mmap [58] is a feature added to the standard UNIX sockets in the Linux Kernel[1], using packet buffers in a memory region shared (mmaped, hence 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 copy is avoided and vice versa.

PacketShader [35] 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. PacketShader 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.

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

Netmap [32] provides zero-copy, kernel bypass, batched I/O and support for multi-queuesing. 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, i40e, r8169, forcedeth, e1000, e1000e and still growing) but can emulate its API over any driver at the price of reduced performance.

PF_RING ZC (ZeroCopy) [33] is the combination of ntop's PF_RING and ntop's DNA/LibZero. PF_RING is much like Netmap [60], 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 as 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 3.1 but no further testing. Anyway, the results of our work should be applicable to PF_RING DNA/ZC.

DPDK. The Data Plane Development Kit[31] is somehow comparable to Netmap and PF_RING ZC, but provides more userlevel functionalities such as a multi-core framework with enhanced NUMA-awareness, and libraries for packet manipulation across cores. DPDK also provides two execution models: 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.

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

OpenOnload [34] is comparable to DPDK but made by SolarFlare, only for their products. OpenOnload also includes a userlevel network stack to accelerate existing applications.

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

Table 2.1 summarize the features of the I/O frameworks we consider.

2.2.3 The case of Netmap and DPDK

From all of those, Netmap and DPDK are probably the most known and used ones.

Netmap uses shadow NIC rings visible to both kernel and userspace as a way to abstract hardware specifics (top of fig. 2.3). When the interrupt is received from the NIC, the interrupt handler of Netmap only sets some flags and wakes up the application (if it was sleeping until packets are available), but does not process any packet. Even under high interrupt storm, this does not hog the CPU at all. All the receive-path work is deferred to the application. The application must use a select-like operation or an ioctl to synchronize the transmission state of the shadow ring descriptors with the real rings used by the NIC. The select operation will sleep until the receive interrupt arrives, while the ioctl will return directly. We refer to both as the *synchronization operation*. The synchronization operation does not actually return any information, it only updates the shadow ring state so the user can access the ring and look if some packets are available.

Netmap avoids the 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, the same region also used by the shadow rings. But the shadow rings come at the price of a higher memory usage and prevents accessing NIC specific features such as reading the RSS hash or VLAN tag of the packet when hardware offloading is enabled without core modifications to the Netmap module.

The transmit path is not shown in figure 2.3, but the logic is only inverted. The applications put the packets ready for transmission in the transmit shadow ring. At this point Netmap has no way to actually know if it should synchronize the state of the NIC ring with the shadow ring. The application must call the synchronization operation

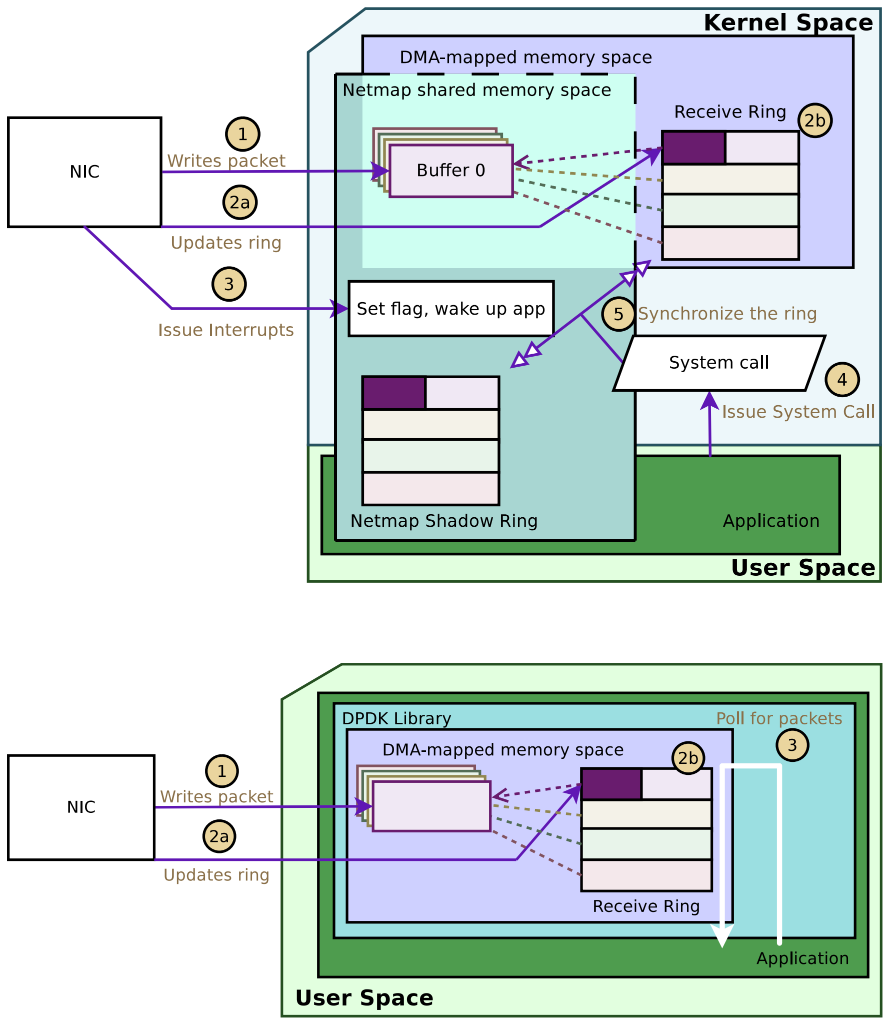


Figure 2.3: Netmap (top) and DPDK (bottom) inner workings. Netmap uses a shadow ring to allow the userspace application to receive packets residing in a shared memory region. DPDK implements the driver in userspace, the application therefore have direct access to the ring used by the NIC and the buffers.

(the select or the ioctl) to tell Netmap that some packets were placed in the shadow ring and that it should synchronize its state with the real rings used by the NIC.

To avoid the need for some common structure hiding hardware and driver specifics, DPDK implements the drivers directly in userspace [1] (bottom of fig. 2.3). The ring used by the NIC is directly the one residing in userspace memory. This is enabled by some kernel facility only invoked at initialization time. As DPDK disables interruptions, the application must constantly loop over the ring (bottom of fig. 2.3 - 3) to check if packets are available. This is known as *polling*. This leads to a higher maintenance cost for DPDK as its developers must maintain a full set of userspace drivers whereas Netmap developers only needs to patch kernel drivers to hook in the interrupt receive loop to prevent the normal path from being called, keeping all of the driver's logic for device initialization.

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 [61]. The motherboard is an Asus P9X79-E WS[62] with 4×4 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 linerate, even with the tools from framework's author [32, 35]. Our experiments lead to the same conclusion, our system seems to be capped at 33 Gbps with 64-byte packets.

To be sure that differences in performances are due to changes in the tested platform, a Tilera TileENCORE Gx36 card fully able to reach linerate in both receive and transmit side was used. We used a little software of our own available at [63] to generate packets on the Tilera at linerate towards the computer running the tested framework connected with 4 SFP+ DirectAttach cables. The generator counts the number of packets received back. All throughput measurements later in this chapter indicates 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 the

[1]Some DPDK non-Intel hardware use bifurcated driver, keeping most logic inside the kernel and allowing operation through standard APIs like ethtool, but exposing buffer memory to userspace like any other DPDK driver.

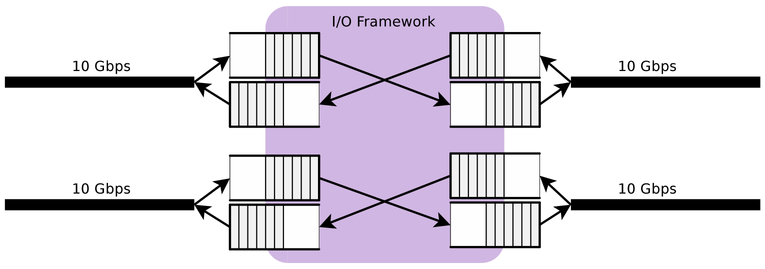


Figure 2.4: Forwarding test case. Packets are directly transferred to the opposite link

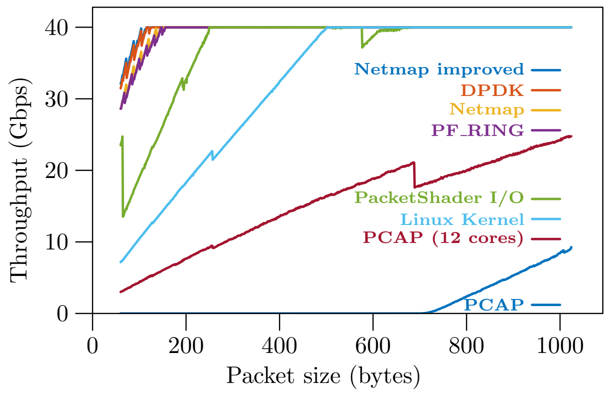


Figure 2.5: Forwarding throughput for some I/O frameworks using 4 cores and no multi-queueing.

number of packets received back after 3 seconds to let the server reach a stable state and compute the throughput after 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 to a pre-defined, hardwired output device as shown on figure 2.4. Each framework has been tuned for its best working configuration including batch size, IRQ affinity, number of cores, thread-pinning and multi-queues. 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.

We realized that our NICs have a feature whereby the status of a transmit packet ring can be mirrored in memory at a lower performance cost than the original method, directly accessing the NIC register through the PCIe bus. We modified Netmap[1] to exploit this feature and also limited the release rate of packets consumed by the NIC to only recover buffers for sent packets once every interrupt, which released the PCIe bus of useless information and brought Netmap performances above DPDK as we can see in figure 2.5. For 64-byte packets, these improvements boost the throughput by 14%, 1.5% over DPDK. However, except for the line labelled "Netmap Improved" in figure 2.5, only the final evaluation in section 3.3 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 2.5.

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 disappear and performance is a little better. This leads us to think that the bottleneck is not on the CPU or memory side, as from their point of view, having 4 NICs on 2 or 4 PCIe does not change a thing. We believe them to be a per-transfer bottleneck in the NIC or in the PCIe bus, as when the size increase less than 16 bytes, the performance increases, but when adding the 16th byte, a drop appears as if a new chunk of data was needed. However PCIe word size is 32 bits, while its theorical throughput (PCIe 2.0 x8) is $32 Gbits/s$ which should be enough to handle the $\sim 16 Gbits/s$ of link layer payload a dual 10G NIC represents when transferring minimal-size packets, given that the inter-frame, CRC, preamble and start of frame are not copied. Therefore we doubt it's linked to the PCIe bus, and probably some limits in the NIC itself, but we are not able to prove it. 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 copy to userspace, while the other frameworks that use zero-copy do not even touch the packet content in these tests and do not pay the price of a memory fetch.

We also made a little Linux module available at [64] which transmits all received packets on an interface back to the same interface directly within the interrupt context.

[1]The usage of the transmit ring status is now merged in mainline Netmap, but not the possibility to ask for non-synchronizing transmission, that is just telling the NIC that packets are available for transmission but not recover sent buffers

The module should show the better performances that the kernel is able to provide in the same 4 cores and no multi-queues conditions. The relative slowness of the module compared to the other frameworks can be explained by the fact the receive path involves the building of the heavy sk_buff and the whole path involves multiple locking, even if in our case no device is shared between multiple cores. This important result shows that as is, the Linux Kernel, even doing nothing (the packets are untouched in this scenario) is not able to achieve high-speed. Therefore, we'll try to tackle some of those problems in section 2.4.

PCAP (that relies on PACKET_MMAP) shows very poor performance because it does not bypass the kernel like the other frameworks and, in addition to the processing explained for the Linux module, packet need to go through the kernel network stack to find its path to the PCAP application. But the bigger problem is that PCAP relies on the kernel to get packets, and each IRQ causes too much processing by the kernel, which is overwhelming a single core and does not let enough time for the userlevel application to actually consume the packets, even with NAPI and IRQ throttling enabled as already discussed in section 2.2.1. The solution is either to use a polling system like in FreeBSD [65] 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-queues and RSS to ensure that each core receives fewer packets than the critical livelock threshold. Our kernel module is not subject to the receive livelock problem because the forwarding of the packet is handled in the IRQ routine which does not interrupt itself.

To circumvent the livelock problem seen with PCAP, we used the 12 hyper-threads available on our platform and 2 hardware queue per NICs to dispatch interrupt requests on the first 8 logical cores, while 4 PCAP threads forward packets on the remaining 4 logical cores. The line labeled "PCAP (12 cores)" gave the best results we could achieve out of many possible configurations allocating diverse number of cores to serve the IRQ and the PCAP threads.

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

2.4 An attempt at fixing the kernel I/O limits

Some packet processing software run in-kernel like the Click Modular Router[11] or its enhanced version for multi-queueing RouteBricks[15]. At the price of harder development and many precautions to be taken, running applications in-kernel obviously avoids the need to copy the packet content to a userspace buffer. But even in-kernel applications that need to receive all packets from a given device without the network stack overhead offer a very limited throughput as discussed in section 2.3. This section analyses why, and tries to propose partial solutions.

From the analysis in section 2.1 can be derived two main observations that could be applied in-kernel:

The kernel application needs to apply backpressure. Both Netmap and DPDK will actually handle the packets when the userspace application calls the framework. Therefore when the NIC is dedicated to one application, the interrupt stops processing packets as soon as the application is stalling. Instead, the kernel reads packets, process them, fill their sk_buff, only to drop them afterwards. Possibly even stalling the application even more. The most common way to hook into the Kernel receive path to intercept all packets from a specific NIC is to use the rx_handler feature. The rx_handler allows to register a function that will be called when the drivers pass the packet to the network stack. However at this point the driver has already done much work such as initializing the sk_buff. Applications that install a rx_handler will receive all packets from that device and can return some value to allow the packets to continue its traversal through the network stack (RX_HANDLER_PASS) or tell that it was consumed and the driver should process the next packet in line with RX_HANDLER_CONSUMED. The in-kernel version of the Click Modular Router uses the rx_handler facility. We modified the Linux Kernel 4.4.77 to add a return value similar to CONSUMED, but indicates that the device should stop processing packets until further notice (RX_HANDLER_DROPPED).

If the device was shared between multiple applications, the application taking full possession of the flow control of the device would be a problem. In which case, the backpressure would block receiving and would stall other applications. However, programs such as Click may take all packets and re-injects the one that needs to go to the normal network stack at a later stage. Recent NICs also support multiple receive queues, using some filters to direct different kinds of traffic to different queues. This allows to use different queues for different applications, *e.g.* the HTTP traffic would go to one specific queue. The backpressure can then be propagated by an application only to its own receive queues.

Normally, drivers using the NAPI (all high-speed drivers as of today) do not handle packets in the interrupt loop anymore but schedule NAPI and mask interrupts. The NAPI polling function will then run, process a certain *budget* of packets from the receive ring, and then re-enable interrupts.

With our modification, when a driver is informed that pressure should be applied by passing the new return values from functions to functions, the driver can stop emptying the receive ring right away. But in itself stopping the polling function is not sufficient as the interrupt will be un-masked when the NAPI function returns. The interrupt request will then happen very fast if the throughput is still high, and the driver will re-schedule the NAPI polling function right away. This would only giving a few more cycles to the application to actually consume packets, if any. Therefore in that case we also don't re-enable interruption, and expose a generic function that drivers must implement to re-enable interruption when the application has processed some packets. The work needed in the driver is minimal: only break the loop when the DROPPED value is returned, and move the code to re-enable the interruption to an exposed function.

A proof-of-concept, still perfectible but working version of the Linux Kernel 4.4.77 is available at [66]. The patch adds the rx_handler return code and implements the driver handling for the i40e driver, but the technique should apply to most drivers easily. The modification in the i40e driver only involves adding 59 lines, moving the re-enabling of IRQ out of the NAPI polling function so it can be delayed if pressure needs to be applied. We modified the in-kernel Click Modular Router to use that functionality, Click re-enables interruptions when the queue is being cleaned bellow 1/3 of its capacity[67].

To test the throughput gain, we pinned the interrupt and the application to the same cores, hence the application cannot process any packet when the rate is too high as already shown in section 2.3. The pressure system reduces unnecessary processing in the receive loop and handle more load. The backpressure proof-of-concept allows handling 9.9**G**Bps throughput forwarding 64bytes UDP packets using a single core serving a 40G NIC instead of the 4.3**M**bps achieved in the same situation without back-pressure.

4.3**M**bps may seam unrealistically low. When there are multiple CPU cores available to serve the interrupts on a dedicated core, Linux will balance the application and the interrupts so this low throughput would not happen in practice. Nevertheless, this state of affairs consumes a lot of CPU cycles to process packets that will be dropped before reaching the application. Also, while CPU are nearly always multi-cores nowadays, some virtual machines may run with a single core. The guest operating system would therefore not be able to balance the interrupt and the application on different cores, leaving no time for the application to run while processing interrupts, as in this test case. This test uses UDP packets. With TCP, the sender would eventually decrease the sending rate when packets are dropped. However with many active connections, or under a UDP or TCP SYN flood attack, the livelock problem would also appear.

Kernel applications need to hook in the receive path before the heavy sk_buff creation. Specific in-kernel applications like Click or the *OpenFlow Virtual Switch (OVS)* will:

A. handle completely the networking stack by themselves including the layer 2

B. not need most of the kernel sk_buff fields and its lot of pre-defined variables for handling of multiple protocols and sockets

Those applications only need to receive a buffer pointer and a length to process a packet. Therefore a generic way to hook right in the driver would benefit such use case. A very recent initiative, the *eXpress Data Path (XDP)* allows hooking into the driver before the sk_buff creation. However in its current state, XDP relies on an eBPF program. XDP does not allow to pass the packet to a third party x86 application (either in-kernel or in userspace) and is limited by the instruction set of eBPF[68].

Figure 2.6 shows the performance of a packet counter (which also touches the packet) using the rx_handler facility to receive packets, compared to using a function inserted into the XDP handler instead of a BPF program. Indeed the BPF program structure is composed of a pointer to a function to be called to execute the BPF code and an array of BPF instructions. The function pointer can be changed by any kernel module to a normal function. This shows the potential of a proper system to hook closer to the driver. At the time of writing, the transmission possibilities of XDP are still limited but there are plans for full support for transmission between devices using XDP.

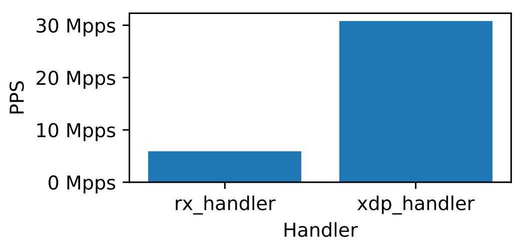
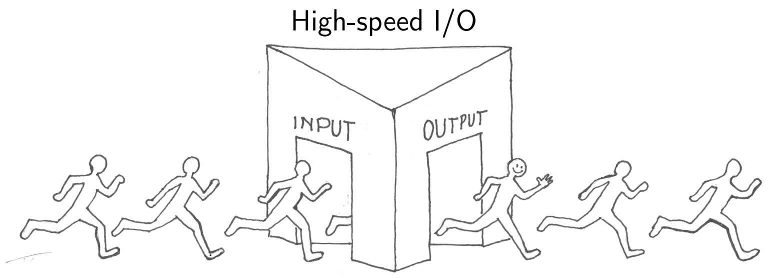


Figure 2.6: Counting the amount of 64bytes UDP packets received using the rx_handler facility or a hijacked xdp_handler running a normal module functions instead of a BPF program. MAC addresses are rewritten to actually touch the packets. Two 40G NICs with one queue pinned to the same unique core.

Note that XDP does not solve backpressure, both are complementary. On top of those changes, the Kernel could also allow I/O batching, but changes from the two observations would already put kernel fast data-path much more on a par with the userspace I/O frameworks.

These techniques, however, do not solve the userspace copy problem. Packet_mmap shares buffers with userspace, but the problem is that a copy from the NIC DMA driver to that space still needs to be made inside the kernel. Therefore the copy is just shifted to another place. Providing a generic way for drivers to allow their buffers to be shared with an application would represent much more work. However, some initiative (probably pushed by the rise of all the userspace I/O frameworks) like [69] want to allow this possibility but are not yet ready.



3

A high-speed packet processing platform

Chapter 2 conducted a review of most known high-speed I/O frameworks and showed that recent frameworks such as Netmap[32] and the Data Plane Development Kit[31] are more or less on a par, allowing to receive packets at rates over 10 Gbps with a single CPU core.

The issue of software packet processing is now reconsidered, 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.

To explore their performance in a general purpose environment, we then compare the existing off-the-shelf integrations of some of these frameworks in the Click Modular Router[11].

Click enables programmers to build routers by composing graphs of *elements*, each executing 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 has already 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 and NFV (even if it is mostly router-oriented for now).

3. A HIGH-SPEED PACKET PROCESSING PLATFORM

Multiple authors proposed enhancements to the Click Modular Router. RouteBricks[15] focuses on exploiting parallelism and was one of the first to use the multi-queue support of recent NICs for that purpose. However, RouteBricks only supports the in-kernel version of Click. DoubleClick [12] focuses on batching to improve overall performances of Click with PacketShader I/O[35]. SNAP[70] 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 piece of work provide useful tips and improvements for enhancing Click, and more generally building an application on top of a “raw packet” interface.

The first part of this chapter conducts a critical analysis of those enhancements, and discusses how they interact with each other and with userspace I/O frameworks, both from a performance and ease of configuration 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 systematic survey of their performance and, more importantly, interactions between the features they provide.

Our contributions 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, in fact 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, we propose a new userspace I/O integration in Click (including a reworked integration of Netmap, and a novel integration of 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. Contrary to previous work which broke 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 is never mandatory.

Section 3.1 discusses how some of the frameworks presented in chapter 2 were integrated into the Click modular router. Section 3.2 analyses those integrations, and various improvements to Click, giving insights into the design of fast userspace packet processors. We then propose FastClick, based on what we learned. Finally, section 3.3 evaluates the performance of our implementation.

This chapter in a nutshell

▶ **Context of this chapter**

- Multiple techniques and parameters need to be chosen carefully to enable fast packet processing on top of I/O frameworks.
- There are multiple integrations of those techniques inside Click, focusing on specific features such as I/O batching or multi-queueing but most often not the interactions between them.

▶ **Highlight of our main contributions in this chapter**

We build FastClick, an enhanced version of the Click Modular Router powered by the following contributions:

- We conduct a review of features enabling high-speed I/O, how to integrate them and for the first time at this scale, carefully decouple them and study their interactions.
- We show the pull path of Click is now unneeded and speed can be gained from using a full-push path.
- We propose a batching implementation and study methods for backward compatibility, enabling the migration of complex functions from simple per-packet processing.
- Beyond network I/O, other system calls impact performance in userlevel. We built a userlevel Clock for x86 systems that is much faster than relying on the OS facilities without losing precision.

■

3.1 A modular high-speed packet processing platform

Contribution notice

Most of sections 3.1, 3.2 and the above introduction are from published work [54] made in collaboration with Cyril Soldani and Laurent Mathy. Section 4.2 is also based on this article but has been largely reworked.

One man cannot solve all the world's problems ■

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.

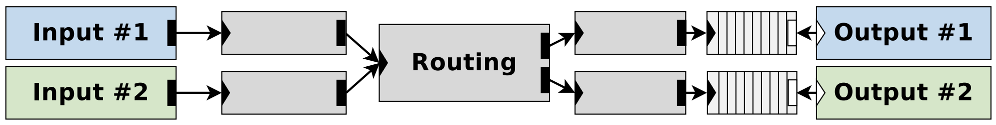


Figure 3.1: Example of a usual path of the Click Modular Router.

In Click, packet processors are built as a set of interconnected processing elements (fig 3.1. 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 it is not interrupted and will process a certain number of packets one by one through the downstream Click elements and give back control to the scheduler only when it has finished. 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 software queues.

Each Click thread runs a loop of 3 actions: the thread executes its assigned set of **tasks**, executes expired software-based **timers** and then uses a select mechanism to sleep on some file descriptors until some **event** occurs as depicted in figure 3.2.

At each round, Click's thread runs up to a configurable limit of 128 **tasks**. //, potentially multiple times the same tasks. Generally, tasks process some data (packets from a device or a software queue) and unschedule themselves when there is no more data to process; the 128 tasks limit is rarely reached in practice.

Timers are purely software based. After processing the tasks, the thread will loop through a list of timers structures and check if the time set to fire some functions has expired.

Tasks may register a set of file descriptors for Click to wait on, using the select mechanism after the timers executed. Generally, when an I/O element has no more input, it will unschedule its task and register a file descriptor that will enable the Click thread to sleep until an **event** occurs. Most of the time the event is the availability of data. The select mechanism allows to set a timeout to wait for some maximal amount of time. It is set to 0 if there are pending tasks, or at the time of the next

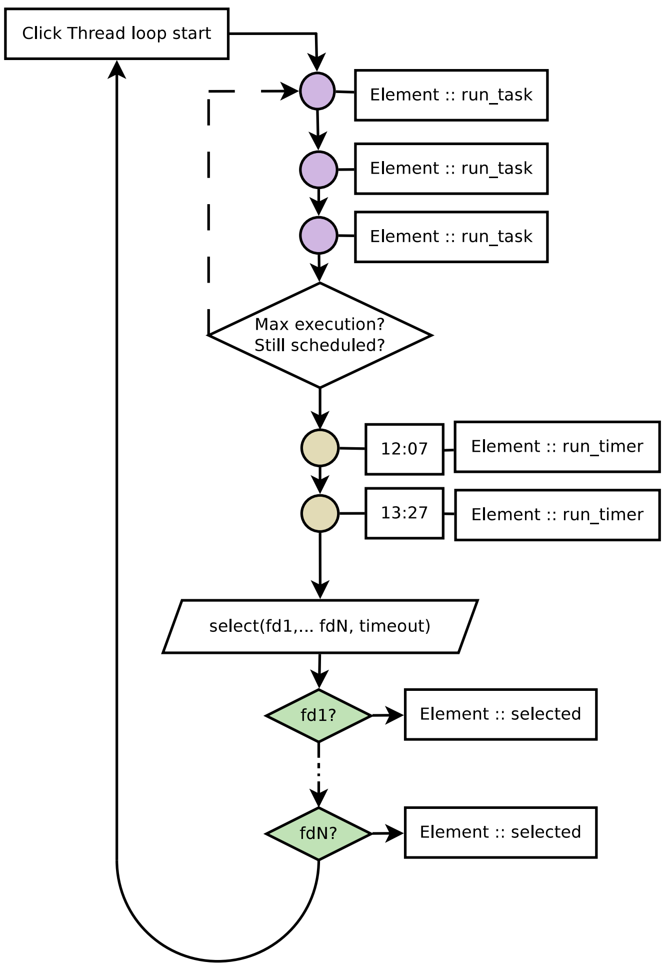


Figure 3.2: Execution loop of a Click thread.

timer to expire. When the select returns specifying that some file descriptors have data available for processing, a function from the element associated to the file descriptor will be executed. In general that function will only remove the file descriptor from the file descriptor list and re-schedule the element's task that will run at the next thread loop.

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 device), the vanilla Click task model forces packets to be processed individually by each element, with parallelism resulting from the chaining of several tasks interconnected by software queues.

Also, vanilla Click uses its own packet pool, copying each packet to and from the buffers used to communicate with the interface (or hardware queues). As such, even if the I/O framework supports zero-copy, vanilla Click in userlevel imposes two memory-to-memory copies (one at the input and one at the output).

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 read or write system call.

- 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[12]: 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-queueing.

- Kernel Click: To demonstrate the case for *userspace* packet processing, we also run the kernel-mode Click. We only modified Kernel Click to support the reception of interrupts to multiple cores. Interrupt processing (creating a sk_buff for each incoming packets) is very costly and using multiple hardware queues pinned to different cores spreads the work. Our modification has been merged in the mainline Click[71].

  Kernel Click had a patch for polling mode, but this is not used because it only works for the e1000 driver and only supports very old kernels which prevent our system from running correctly.

	Netmap	PCAP	UNIX Sockets	DoubleClick	Kernel	FastClick Netmap	FastClick DPDK
IO Framework	Netmap	PCAP	Linux socket	PSIO	Linux Kernel	Netmap	DPDK
IO Batching	Y	N	N	Y	N	Y	Y
Computation Batching	N	N	N	Y	N	Y	Y
Multi-queue support	Y	N	N	Y	N	Y	Y
No copy inside Click	N	N	N	Y	N	Y	Y

Table 3.1: Click integrations of I/O frameworks.

Table 3.1 summarizes the features of these I/O framework integrations into Click.

We ran tests for pure packet forwarding, similar to those in section 2.2.2, 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 software 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 the 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 integrations, FromDevice and ToDevice are pinned to the same core. The results are shown in figure 3.3. The top two lines, labelled FastClick, should be ignored for the moment. 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 livelock behaviour as the one observed in section 2.3. The same configurations where interrupt requests (IRQ) are dispatched to 8 logical cores and 4 logical cores are kept 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 livelock for the same reason as 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 sk_buff 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 software queue while nearly never servicing the queue consumer.

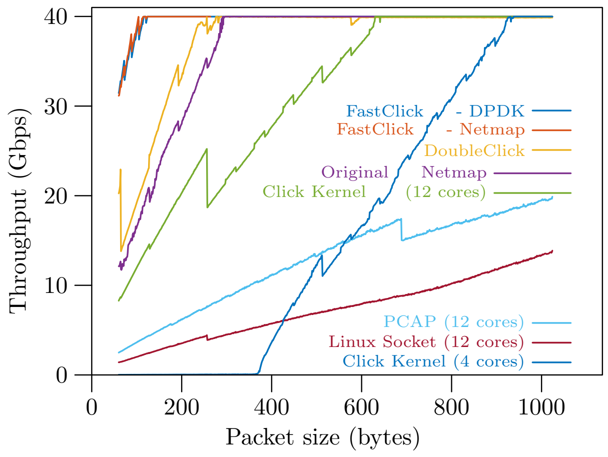


Figure 3.3: Forwarding throughput for some Click I/O implementations with multiples I/O systems using 4 cores except for integration heavily subject to receive livelock.

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 element. Each interface represents a different subnetwork, and traffic is generated so that each interface receives packets destined equally to the 3 other interfaces, to ensure we can reach linerate 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. 3.4). There is a large probability of having very small flows (16% of the flows will have between 1 and 16 packets), a small probability of having middle-sized flows (6% of packet will have between 30 and 80 packets), and a small probability of having long flows (7% of packet will have between 96 and 128 packets) to reproduce the observed "mice and elephants" phenomenon seen in the Internet[72].

Results are shown in figure 3.5. We omit the PCAP and socket modes as their performance is very low in the forwarding test. Additionally, we show the Linux kernel

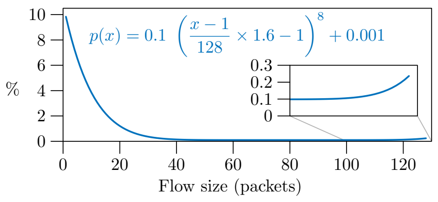


Figure 3.4: Probability of having flows of 1 to 128 packets for the router packet generator.

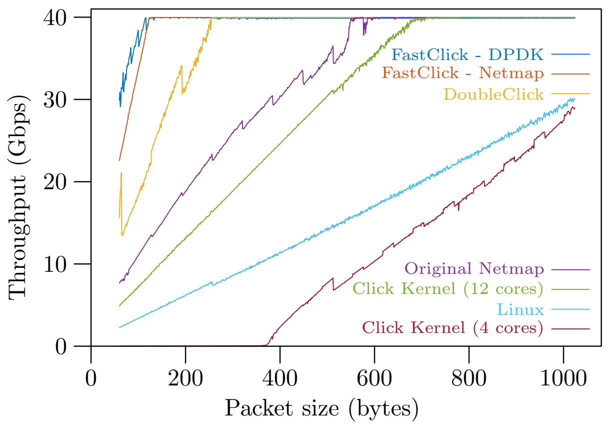


Figure 3.5: Throughput in router configuration using 4 cores except for in-kernel Click.

routing functionality as a reference point. Again, ignore the lines labelled FastClick for now.

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 the Kernel does much more processing to build the sk_buffs 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 livelock 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.

3.2 I/O analysis

We now present an in-depth analysis of the features used by the Click integrations studied in section 3.1, 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. See section 2.2.3 for a more detailed walk-through of Netmap and DPDK.

The following section starts from vanilla Click “as is”. Features will be reviewed and added one by one.

3.2.1 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 3.6 assesses the impact of I/O batching using the vanilla Click Netmap integration in a modified version to force a synchronization call after multiple packets are received up to a certain batch size in both input and output. As explained in more depth in section 2.2.2, the synchronization call is the facility provided by the underlying I/O framework to check if some packets are available for processing or tell that packets are waiting to be sent. With DPDK, the synchronization is done after some packets are passed to DPDK through the DPDK API, however Netmap has no way to know if some packets were added in the transmit ring which is simply memory mapped and thus relies the use of a select-like[1] facility or a specific ioctl.

Vanilla Click always processes all available packets before calling again Netmap's select method – the select system call asks Netmap to synchronize the device input ring with Netmap's shadow rings and updates ring indices allowing to compute how many packets are in the input ring. Click reads available packets in batches and transmits them 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 of synched data is available. When not rescheduling, both methods go back to the select system call that will synchronize the hardware and shadow rings and will re-schedule the tasks when the select indicates some data is available. This sequence of operations will probably enable a bigger batch at the next run as Netmap will have synchronized with packets received by the NIC while Click was processing the last batch through the graph. This strategy tends to force the use of bigger batches and thus preserves the advantages of I/O batching.

On the other hand, the ToDevice completely fills the Netmap output ring before synchronizing (*i.e.* flushing) it, that is to declare these packets as available for sending to the output NIC. From that moment, the ToDevice uses a select-like facility to check for available space in the output ring. However, that select will return even if a few packets have actually been transmitted, releasing the ToDevice to fetch more packets from the Click software queue to fill the Netmap output ring again. This thus results in a short sequence of synchronize operations on the Netmap output ring, cancelling the batching effect (as the number of synchronize per packet shoots up), resulting in higher overhead. In essence, the first synchronize simply happens too late and the Netmap output ring never really gets a chance to empty. The resulting high number of synchronization call to Netmap reduces the TX throughput and force the interconnect

[1]The select operation in Linux allows a thread to wait on some file descriptors until some data is available. That is for Netmap, wait until one NIC specific ring has received some packets. In practice, Click uses the poll method that is more flexible than select and most of the time faster. Poll is not to be mixed up with the concept of polling as referred to when speaking of DPDK. DPDK drivers have no select-like mechanism and therefore rely on the CPU looping over the NIC ring checking if some packets are available. To avoid confusion we'll refer to the Linux poll method as select and poll will always refer as the polling method as implemented by DPDK.

Queue between the FromDevice and the ToDevice to start buffering packets up to a point where packets are dropped.

As a solution, we propose to synchronize the Netmap output ring much sooner, that is when the number of unsynchronized packets in the shadow output ring reaches a smaller batch limit, or when the input dries out. Our ToDevice constructs *internally* a batch of the desired size, and dumps it into the Netmap output queue in one go, and call the synchronize operation afterwards.

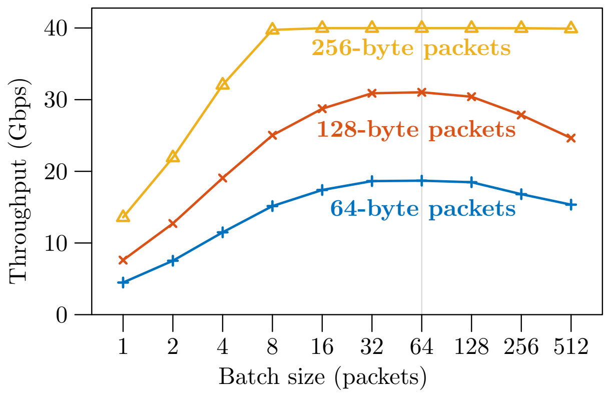


Figure 3.6: I/O Batching – Vanilla Click using Netmap with 4 cores using the forwarding test case (queue size = 512). A synchronization is forced after each "batch size".

As DPDK disable interruptions[1], the input task is running in loops, never stopping and eating 100% of the CPU. The difference with Netmap is that as it does not support a select mechanism for the transmit side, no pull path is possible. A simulated one using the full CPU, or relying on a timer could be implemented. However for reasons that will be further explained in section 3.2.3, we implemented the push approach only. When packets are pushed to the DPDK ToDevice, they are kept in an internal software queue up to a certain batch size. When the threshold is reached, packets are passed to DPDK for transmission. To prevent waiting for too long while the batch grows to the given size, a timer is also set to flush the packets when they are queued. In Click, timers run after all tasks and there is no interruption. Therefore the default timeout value is 0 seconds, meaning that after all tasks have run, the output internal queues will be flushed as the timeout will have expired directly.

[1]DPDK introduced recently support for re-enabling interruption for the receive side but we did not add support yet

The resulting effect is that all FromDevice elements will process available packets in the input hardware queue up to the input "burst" limit through the Click graph. Packets will eventually be queued in the ToDevice's internal queue. If enough packets are queued, they will be transmitted to DPDK directly. The leftovers will be passed to DPDK when the timer expires, that is right after the FromDevice's task finishes. This enables to avoid waiting for a certain time for the ToDevice's internal queue to fill up, which would increase latency when the rate is low.

To evaluate performance impact of the I/O batch size with DPDK in terms of throughput but also in term of latency under realistic conditions, we built a third test-case (fig. 3.7) similar to the router scenario running on a machine which has an E5-2630 v3 CPU, and 32 GB of RAM. We used traffic captured from our University campus at the point of connexion to the Internet. The traffic is kept in two traces, one with the traffic from the LAN to the WAN and one with the traffic from the WAN to the LAN. Both traces are replayed towards two different port of the router as fast as we can, but in relative order (the timing is accelerated but the order of packets between traces is kept). The *replayer* uses FastClick (in its final version) with a configuration to preload the traces in memory and replay them on a computer similar to the *DUT* connected to the same switch. Therefore the *replayer* acts both as the WAN side and the LAN side source and sink to be able to do latency measurements, as packets which are not dropped by the router should arrive on the other side of the *replayer*.

As a single core can generally handle 10G of mixed packet size, we used dual-40G Intel XL710 QDA2 NICs, on the two computers interconnected by a 40G switch. The replayer is able to replay the WAN to LAN (referenced as RX further) side at $\tilde{2}$8G and replays the LAN to WAN (referenced as TX further) side at 12G which is not the limiting one, so we omit it in all graphs. The TX side has a lower throughput, because at the time of capture the TX transmission was lower than the TX transmission. As the relative order between traces is kept, the TX replay side is waiting most of the time.

Results can be seen in figure 3.8. As expected, the input burst size should not be too small (> 8) but has no major impact on performances. The input burst does not matter much as DPDK rely on polling, meaning the FromDevice task will be re-executed quickly. However the output burst size matters more. The throughput will drop especially after an output burst size of 128 packets. This is because the output is idle while the batch is being constructed, it is therefore waiting most of the time.

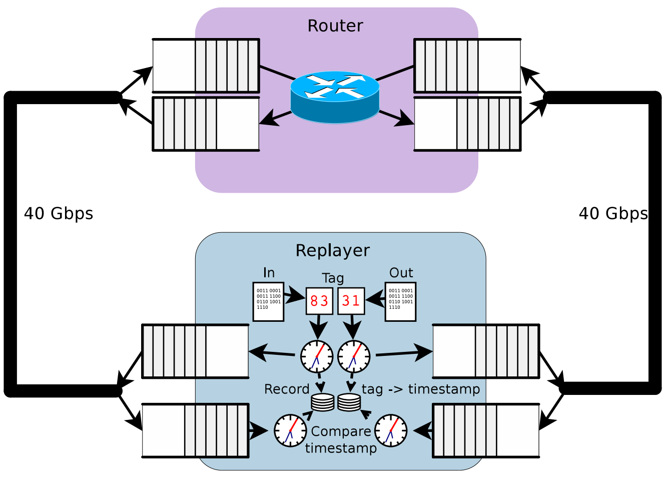


Figure 3.7: Campus test case. The replayer replay traces captured at the University campus point of connection to the internet, tags the packets, associate a timestamp to the tag to compute latency. The two computers are interconnected by a L2 switch, MAC addresses are rewritten to act as in this schema.

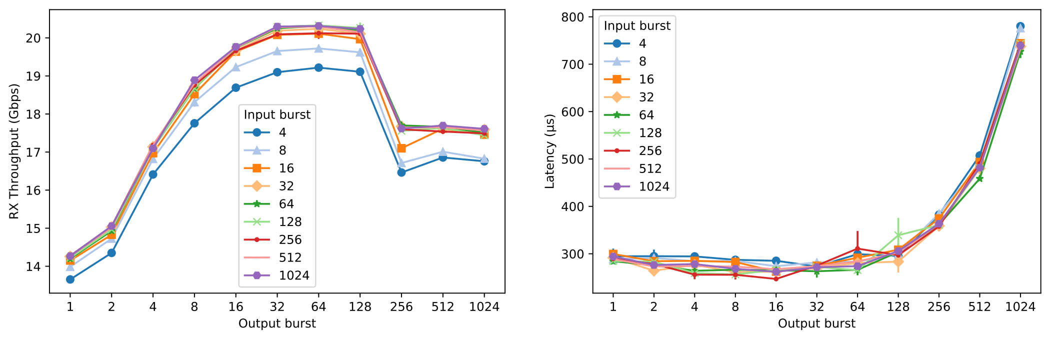


Figure 3.8: I/O Batching - Vanilla Click with our DPDK integration with 1 core using the campus router test case varying input and output burst size. Average of 3 runs per point, standard deviation is shown by error bars (if not visible, runs are stable).

3.2.2 Ring size

The I/O batch size limit is there to ensure that synchronization is not done after too few packets. As such it should not be related to the ring size (the hardware queue size). To convince ourselves, we ran the same test using the forwarding test case and Netmap implementation with multiple ring sizes and found that the better burst choice is more or less independent of the ring size as shown in figure 3.9.

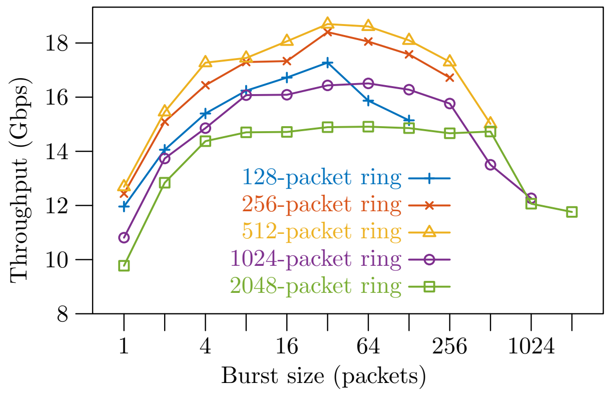


Figure 3.9: Influence of ring size - FastClick using Netmap with 4 cores using the forwarding test case and packets of 64 bytes.

What was surprising though is the influence of the ring size on the performance, especially with Netmap.

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. Indeed, the total number of buffers increase up to a point where all the buffers do not fit in the CPU caches anymore.

As described in section 2.2.2, this is more prominent with Netmap which has a higher memory consumption than DPDK, but the later also shows a drop in performances if using more than 2048 descriptors per ring.

3.2.3 Execution model

In 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 software 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 software queue, traverse the elements between the queue and the ToDevice and are then sent for transmission as shown in figure 3.10 (a).

One advantage of the software 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 enables the ToDevice to receive packets only when it really has space to place packets in the output hardware ring. The ToDevice 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 transferred between cores.

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 3.2.2), so these are sufficient to absorb transient traffic and processing variations, substituting advantageously for the Click software queues.

Therefore, we adopt a model without software queue: the full-push model where packets traverse the whole forwarding path, from FromDevice to ToDevice, without interruption, driven by a single thread. How multiple threads can still be used to accelerate processing of packets from a single NIC in the full-push context will be reviewed in section 4.1.

Packets are taken from the NIC in the FromDevice Element and are pushed one by one into the Click pipe, even if I/O batching is supported. The packet is pushed

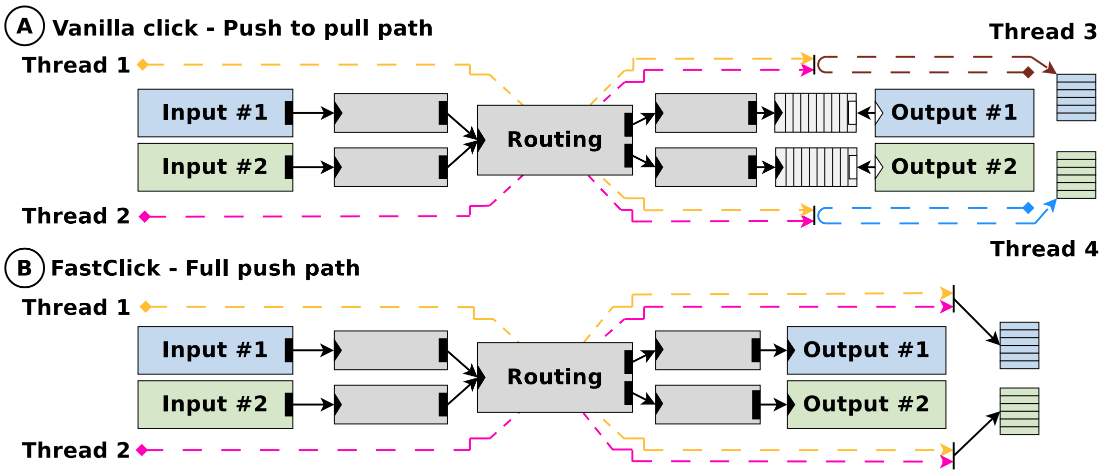


Figure 3.10: Push to Pull and Full Push path in Click.

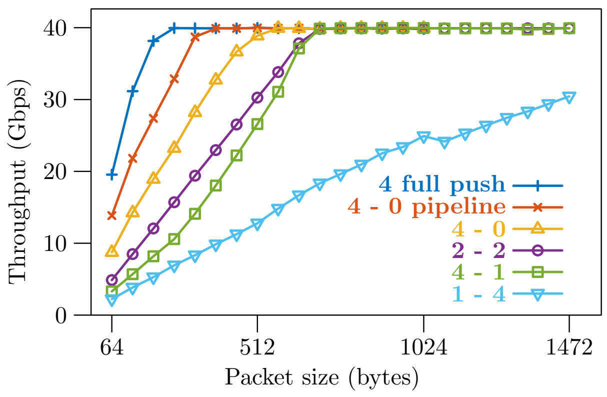


Figure 3.11: Comparison between some execution models. Netmap implementation with 4 or 5 cores using the router test case.

through the pipe until it reaches a ToDevice Element and is added in the transmit buffer as shown in figure 3.10 (b). If there is no empty space in the transmit ring, the thread will either block or discard the packets.

As depicted in figure 3.10, in full push mode, 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. With Netmap the output ring must be synchronized sometime, to tell the NIC that it can send some packets that were placed in the ring. Syncing too often cancels the gain of batching, but syncing too sporadically introduces latency.

When the number 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 new packets can be advertised to the NIC, and available space from packet sent can be reclaimed. This also enables an improvement as the slower select mechanism isn't used anymore for the transmit side, not having to constantly remove and add the Netmap file descriptor to Click's select list. For the DPDK implementation, the ToDevice queues packets into an internal array that can directly be passed to DPDK when it reaches 32 packets as this number proved to be a good performance point for both throughput and latency in section 3.2.1. As already explained in section 3.2.1, we flush packets as soon as the input dries out if 32 packets could not be batched. In both situation we never actually wait for packets to queue up, therefore compared with the pull approach the full-push one does not introduce any latency.

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 3.11 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 different cores running the 4 ToDevice). Therefore the 1-4 line uses 1 core to receive packets and 4 cores to transmit them, while the 4-0 line uses 4 cores to receive packets and schedules the pull path on the same 4 cores. 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.

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 as it was introduced in DoubleClick.

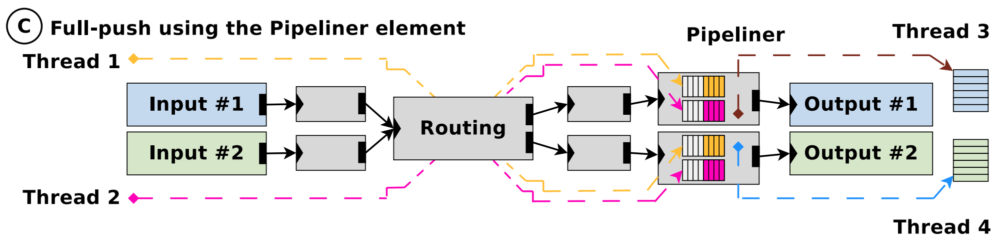


Figure 3.12: Enable queuing in a full-push execution model with the Pipeliner element

3.2.3.1 Pipeliner: a software queue that keeps a full-push path

In vanilla Click, a considerable amount of time is spent in the notification process between the Queue element and the ToDevice. The time spent in synchronization and scheduling reaches up to 60% of CPU time with 64-byte packets for the forwarding test case in push-to-pull mode. We built a new queue that enables queuing but keeping the full-push semantics and advantages, called the Pipeliner element. The Pipeliner schedules a task which will read packets from the queue and push packets downwards. The model is exposed in figure 3.12. Packets pushed to the Pipeliner 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. The performance of the pipeliner is shown in figure 3.11 by the line labeled “4 - 0 pipeline”. Queuing introduce a performance penalty compared to a purely full-push model without queue but performs better than the comparable approach using a thread-safe queue and a push-to-pull path (the “4 - 0” line). Instead of having a pulling thread starting from the last output element constantly adding and removing file descriptors to the select set, the pipeliner enables a much lighter full push path that will only call Netmap’s ioctl from time to time. The per-core queue also enables to remove the atomic operations as single-producer, single-consumer rings can be implemented with normal operations on simple volatile indexes on x86 architectures. Using the technique that will be presented in section 4.2, the Pipeliner knows in advance which threads will push packets to it, and only opens one queue for each thread that can actually reach the Pipeliner.

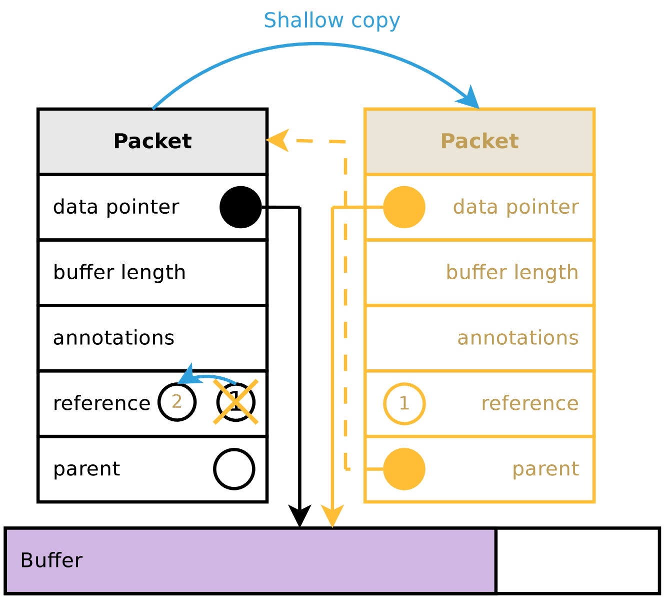


Figure 3.13: Simplified view of a Click packet and its shallow copy

3.2.3.2 Advantage of a full-push configuration for reference counting

In Click, packets can be seen as two parts: one is the packet metadata 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 metadata contains the length of the actual data in the buffer and some annotations to the packet used in the diverse processing functions. Click enables to clone packets using shallow copy. Cloned packets only keep a reference to another packet's buffer, and increments a reference counter for the given buffer (figure 3.13). When cloned packets are destroyed, the reference counter is decremented. The last destroy operation that therefore puts the reference counter to 0 will actually recycle the buffer. In 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 without the new Pipeliner element or any software queue, 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 detects a full push configuration using the technique described in section 4.2.

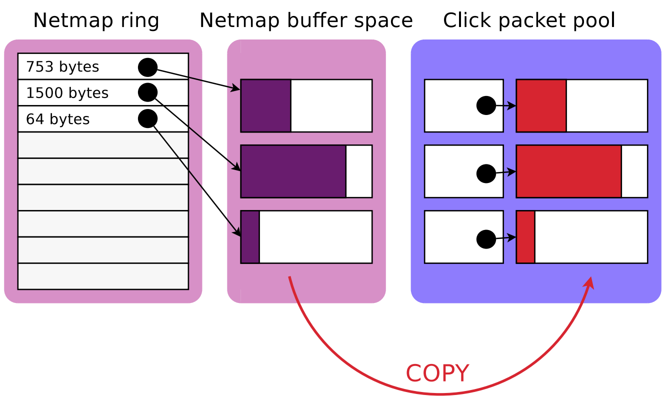


Figure 3.14: Click implementation of the Netmap receive path. Packet content is copied to Click's own buffer.

3.2.4 Zero Copy

In vanilla Click, a buffer space is used 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 metadata 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. This process is shown in figure 3.14.

This packet copy can be avoided when using Netmap, by swapping buffers referenced in the Netmap input and output rings with empty buffers as shown in figure 3.15. 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 packets to arrive. By allocating a number of free buffers and swapping a freshly received packet with a free buffer, we can delay the 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.

Memory referenced in Netmap's ring needs to be DMA-mapped so the NIC can access the memory. We implemented an ioctl using the technique introduced in SNAP [70]

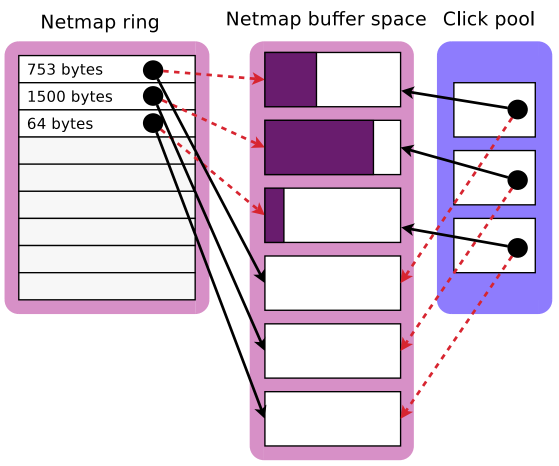


Figure 3.15: FastClick's zero-copy implementation of the Netmap receive path. Dashed red lines are pointer replaced as part of the receive process.

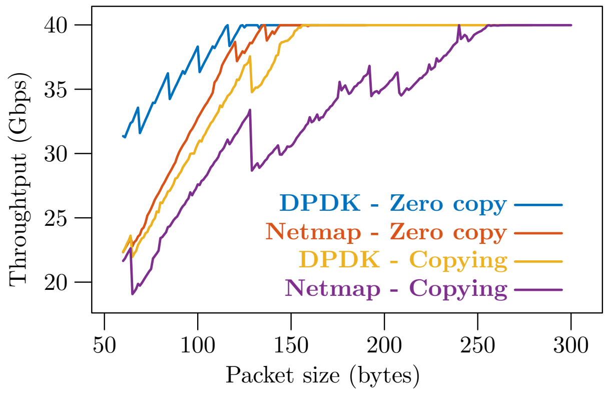


Figure 3.16: Zero copy influence - DPDK and Netmap implementations with 2 cores using the forwarding test case.

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 Netmap buffer pool.

When the Netmap output ring is beginning to be full, the ioctl to sync the output ring is called after each packet to reclaim some buffers. Without zero-copy the buffer copying took some time and ensured that the synchronization was not done too often, actually leaving some time to the NIC to process the packets while the copying is done. With zero-copy, there is no need to copy the buffer content and calling the ioctl after every packets is starting to introduce congestion on the PCIe link[1], and is leading to trashed CPU cycles as the ioctl rely on a system call, which is heavy by nature.

Therefore when the ring is full, the call to the ioctl is disabled until some number of packets are pushed and kept in the internal queue, or when a timer of $1\mu s$ expires, whichever comes first. When the call to the ioctl is re-enabled, it is very likely the Netmap output ring will have sent some packets, and the ioctl will therefore release some space in the output ring.

DPDK directly provides a swapped buffer, as such we do not need to take care of swapping the buffer or copying its content before processing it through the Click graph.

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 3.16. 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.

3.2.5 Multi-queueing

Multi-queueing can be exploited to avoid locking in the full-push paths. Instead of providing one ring for reception and one ring for transmission, 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 3.17.

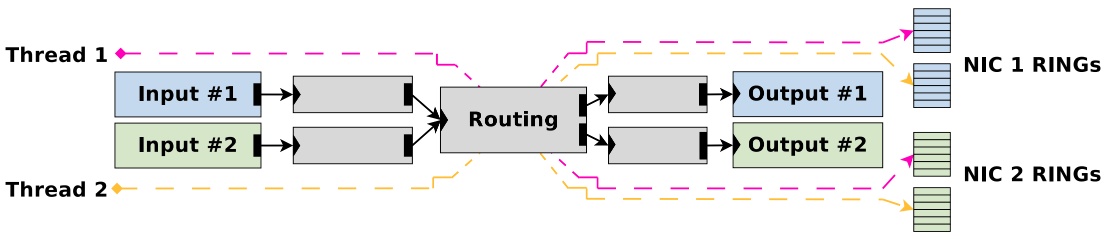


Figure 3.17: Full push path using multi-queue.

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

[1]Actually the synchronization ioctl does not rely on the PCIe bus with out improved Netmap version (see section 2.3). Vanilla Netmap will however directly write to a NIC register instead of relying on the NIC writing the state of the ring in memory and therefore creates PCIe messages when used.

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 3.18.

The evaluation shows that using multiple hardware queues is slower than locking using Netmap, but provides a little improvement with DPDK. With Netmap, increasing the number of queues produces the same results as increasing the number of descriptors per rings, as seen in section 3.2.2. Both end up multiplying the total number of descriptors, increasing the size of Click's working set to the point where it starts to be bigger than our CPU's cache. As zero-copy is used, we see that the cost of reading and writing from and to Netmap's descriptors goes up with the number of queues.

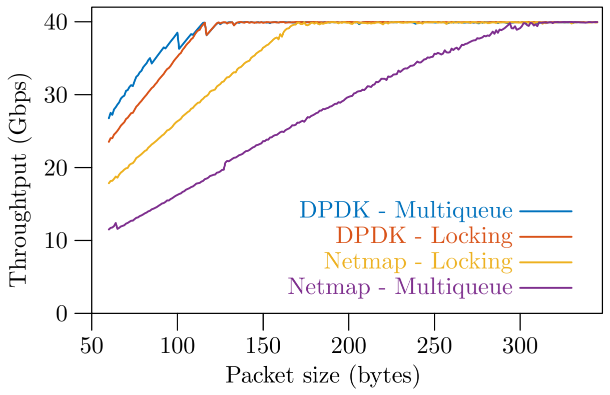


Figure 3.18: Full push path using multi-queue compared to locking - DPDK and Netmap implementations with 4 cores using the router test case.

When using more cores, the cost of locking could increase to the point where multi-queueing becomes more efficient. As Netmap would force us to look at the 12 RX queues when using 12 TX queues, we ran the same experiment with 12 cores using only the DPDK implementation, which can use 3 RX queues per NIC (1 for each input thread) and 12 TX queues (one per output per thread). We also tried to lock per I/O batch, instead of locking per-packet. As it can be seen in figure 3.19, there is no difference between locking and using multi-queueing to avoid contention in the output ring in the case where I/O batching is used. However, without I/O batching, locking indeed becomes more expensive than multi-queueing, due to increased lock contention.

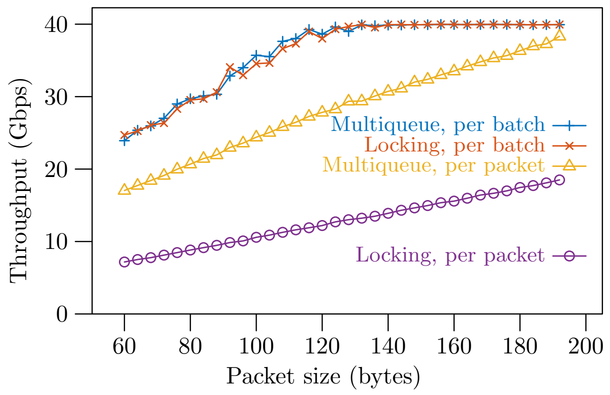


Figure 3.19: Full push path using multi-queue compared to locking - DPDK implementation with 12 cores using the router test case.

3.2.6 Compute Batching

While compute batching is a well-known concept [12, 70] 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 batch would, leading to partially-filled batches. We prefer to use a simply linked list as shown in figure 3.20, for which support is already inside Click, and is better suited to 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 for another data structure. As such, we introduce no new Packet class in Click and more importantly, building batches does not need any memory allocation.

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, enabling to merge batches very efficiently. Indeed, to append a batch to the end of another batch, we only need to add the second batch's head packet as the next packet of the tail of the first batch, and update the count and tail

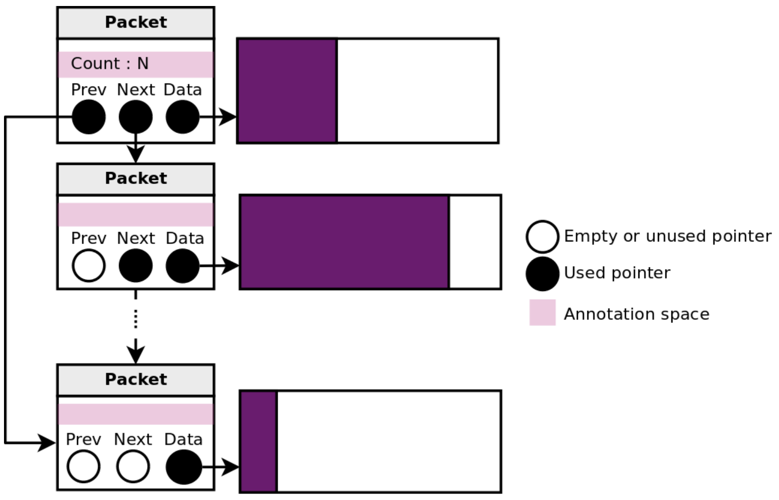


Figure 3.20: Linked list of packets as passed between elements when using compute batching

annotation of the head packet. Merging is therefore a $O(1)$ operation and does not need any memory allocation or release. This is contrasting with vector-based batching where the second batch must be looped through entirely when merging, a $O(N)$ operation where N is the size of the second batch, plus the need to recycle the second vector. It also has the limitation that the total size of the batch must not be bigger than the size of the first vector.

One example of when merging is done is when batches of packets are transmitted and then recycled. The batch of packets needs to be put back in the Click packet pool, which is essentially a long list of packets. A batch of packets that has gone through the router can therefore be recycled in $O(1)$ when using linked list.

3.2.6.1 Batch size

The size of the batch is determined by the number of packets available in the receive ring, up to a chosen burst size limit. We never wait for more packets to ensure a certain minimal size before flushing the batch. It may appear as a contradiction to section 3.2.1 at first glance where we show that small batch sizes reduce performance. What will happen is that when the throughput is high, the batches will grow naturally up to the given batch size limit, therefore ensuring a high throughput and a low latency on average as discussed in section 3.2.1. When the throughput is lower, the batches will become smaller, but as the CPU has also less work to do (as a fact of the lower throughput and therefore fewer packets to handle), the CPU will be able to cope with the overhead introduced by small batch sizes, ensuring a low latency for all packets but still able to cope with the throughput.

This strategy will not pose problems when multiple applications run on the same CPU cores. Indeed, when the rate is low on one application, inducing a little bit overhead on the CPU, either the other applications still have enough CPU share and it is fine, either they do not and will start stalling the first application that, while running, will buffer more packets in the input hardware queue. And therefore when scheduled again will process a bigger batch. Anyway, most high-speed packet processing frameworks encourage or only support the use of one application per CPU core for better performance. Moreover, targeting middlebox applications in this thesis we propose to use dedicated cores for each NIC, because packets from the same input are more likely to follow the same processing and therefore maximise instruction cache hit. CPU core assignation will be discussed in more details in section 4.2.

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, which are the synchronize operation for the output side as explained in section 3.2.1.

With compute batching, a batch of packets is received in the output element and the synchronize 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 synchronize operation will be done more often, reducing latency as packets don't need to be accumulated. While there is still space in the transmit ring, batching also avoids the need to set a timer to flush pending packets if there is no queuing that needs to be done. However a flushing timer is still needed in non-blocking mode where the ToDevice builds an internal batch to cope with sudden burst towards a unique output.

Without batching, a rate-limit mechanism had to be implemented when the ring is full to limit the call to the synchronize operation in full-push mode as explained in section 3.2.4. Indeed, in this case, the synchronize 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 synchronize 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

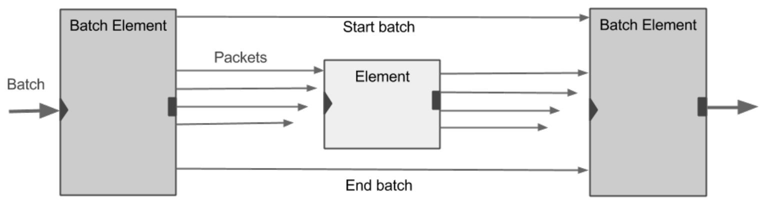


Figure 3.21: Un-batching and re-batching of packets between two batch-compatible elements for a single path

batching is used as the time to process the batch by the Click elements gives time to the NIC to empty part of the output ring.

3.2.6.2 Batch-local variables

In some cases, batching can allow us to remove the multi-thread problem caused by mutable data. For example, the IP Routing element that chooses an output according to the packet's destination IP address will cache the address of the last seen packet and its associated output as there is a high probability that the following packets are for the same destination. In that case, if the element is traversed per multiple threads, there would be a concurrency problem as the cache could be modified by multiple threads. With batches of many packets, one can use a local variable (allocated on the thread's stack) to do the caching of the lookup. The only drawback would be if the last packet of a batch has the same destination address than the first packet of the following batch, as an additional lookup will have to be done.

3.2.6.3 Backward compatibility

In both SNAP and DoubleClick, the batches are totally incompatible with the existing Click elements, and one needs to use either a kind of Batcher/Debatcher element (SNAP) or implement new compatible elements. In our implementation, elements which can take advantage of compute batching inherit from the class BatchElement (which inherit from the common ancestor of all elements "Element").

Before starting the router, FastClick makes a router traversal, and finds BatchElements interleaved with simple Elements. In that case the port between the last BatchElement receiving batches (the left one on figure 3.21) and the vanilla Element will unbatch the packets. The port after the vanilla Element will re-batch the packets as shown in

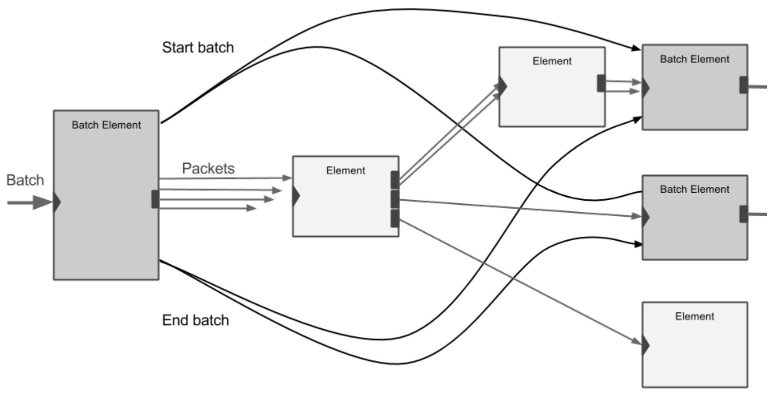


Figure 3.22: Un-batching and re-batching of packets when downstream elements have multiple paths.

figure 3.21. As the port after the vanilla Element cannot know when a batch is finished, the port of the left BatchElement 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 its associated BatchElement. Note that the downstream port uses 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 3.22. This list is also found at configuration time.

While the solution proposed here for backward compatibility is working and provides improvements by keeping the batching semantics, it induces some costs. For elements in the fast path, it is important to implement a proper support for batching as the cost of un-batching and re-batching goes up. 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.

3.2.6.4 Compute batching feature evaluation

Click keeps two pools of objects: one with packet descriptors, and one with packet buffers. SNAP way of handling a freshly available Netmap packet is to take a packet descriptor from the packet descriptor pool and attach the filled Netmap buffer from the receive ring to the packet descriptor. 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 Click buffer pool, getting totally rid of malloc'ed buffers. If FastClick is compiled with Netmap support, the pool of buffers is replaced by a pool of complete Netmap packets, that is a Click Packet descriptor with a Netmap buffer already linked in. Therefore a single pool allocation is needed when receiving Netmap packets. This provides improvement because it is very likely that if Netmap is used, packets will be either received or sent from/to a Netmap device. This is labelled "NPOOL" in figure 3.23.

If there is not enough buffers in the pool, the pool can be expanded by calling the same ioctl than in section 3.2.4 to receive a batch of new Netmap buffers and allocate the corresponding number of descriptors. 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 packet buffers, this is called per-batch recycling and is labelled "RECYC" in figure 3.23.

We do not provide the swapping functionality in our DPDK implementation. As DPDK always swaps the buffer with a free buffer from its internal buffer pool when it receives a packet, we do not need to do it ourselves. We simply use the Click pool to take packet descriptors 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 3.23. 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 determines the packet type by reading its Ethernet type field and dispatches packets to different Click

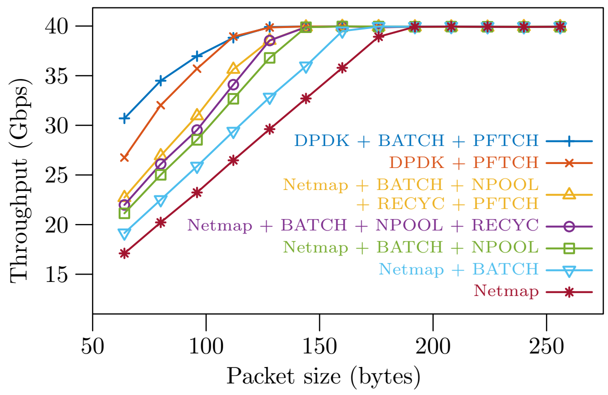


Figure 3.23: Batching evaluation - DPDK and Netmap implementations with 4 cores using the router test case. See section 3.2.6 for more information about acronyms in legend.

paths, the data is already in the cache thanks to prefetching, enabling another small improvement. We omit the forwarding test case because the batching couldn't improve the performance as it doesn't do any processing.

3.3 FastClick evaluation

We repeated the experiments in section 3.1 with our FastClick implementation. The results of the forwarding experiments are shown in figure 3.3, and those of the routing experiment in figure 3.5.

Both Netmap and DPDK FastClick implementations remove the important overhead for small packets, mostly by amortizing the Click wiring cost by using compute 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), because FastClick avoids per-thread duplication of elements. The input elements are auto-configured according to NUMA nodes and available CPUs.

Open Source Availability

FastClick is available at [21].

Try it out ! ■

3.4 Beyond Network I/O

Recent work tackles the matter of removing or highly amortizing system calls for network I/O as depicted in section 3.2. Some works like IX [59] and Arrakis [37] try to get the Operating System out of the way by also leveraging the scheduler and the storage stack for high-speed. However, Kernel programming is still difficult and risky in many regards. As such, high-speed platforms such as VPP[73], NetVM[74], E2[43], OpenBox[41] and many others preferred to keep a purely userspace solution.

Applying most techniques presented in section 3.2 to a large security appliance vendor[1] revealed a bottleneck due to a very high number of calls to the system clock. Multiple VNFs take care of keeping timed statistics about various state of flows, clients, connections, etc. The whole time spent in timing system calls did go up to around 1/5th of the total CPU time.

In this section, we will review a way to avoid relying on time system calls. However others system calls may be problematic, especially with the recent *Kernel Page Table Isolation (KPTI)* patch to counter the Meltdown[75] vulnerability that increase drastically system call cost. For instance, avoiding the kernel to generate *every* pseudo-random numbers or implement lighter message-based inter-process communication could also be the subject of future work.

3.4.1 Timing

Linux kernel and C libraries now use *virtual dynamic shared object (vDSO)* which relies on memory mapping through shared libraries to serve routines that do not specifically need to run in kernel space. However, as shown in figure 3.24, simply time-stamping every packet passing by (which is realistic according to observations of the live security appliance) in the campus router test case induces a performance drop of $\sim 18\%$ even if using vDSO.

On x86 processors, the *Time Stamp Counter (TSC)* is a counter that increase at the frequency of the CPU. On recent CPUs, the frequency of the TSC is fixed, meaning that even if the frequency of some CPU cores is changing for power saving, the TSC will still increment at the same frequency. While the exact frequency of the TSC is unknown, it

[1]Tom Barbette completed a 3 months research internship at Cisco Meraki, trying FastClick techniques in some of their products

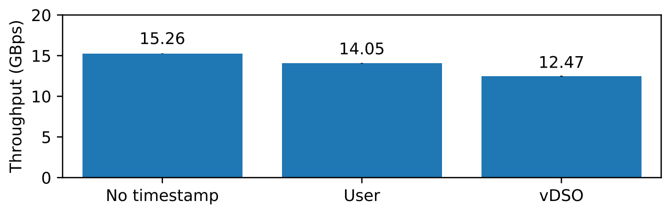


Figure 3.24: Receive side of the campus router test case using one CPU core limited at 1200MHZ, timestamping every packet using clock_gettime(), using our userland TSC-based approach or without any timestamping.

can be approximated. One technique will be described later in this section. The RDTSC instruction allows to read the counter into registers and is not a privileged instruction, meaning it can be used in userlevel. Provided that the TSC is stable enough, it can be used to compute a delta of time using an algorithm like algorithm 1. This is how most Operating Systems compute the time when the TSC is known to be stable.

Most systems also dispose of a High Resolution timer (HPET) which has a more accurate and known-in-advance tick frequency. The value of the timer is read through memory-mapped IO for efficient access, but still more expensive than issuing a RDTSC instruction[76]. Moreover, it is in general around 10Mhz and does not provide a very high precision as the maximum precision would be 100ns. For those reasons the Linux Kernel use the TSC by default even when HPET is available.

Algorithm 1 Time computation using the TSC

```
function INIT
    $T_0 = current\ time$
    $TSC_0 = RDTSC$
end function
function NOW
    $TSC_1 = RDTSC$
    $T_1 = T_0 + (TSC_1 - TSC_0)/TSC\_FREQ$
    or
    $T_1 = T_0 + (TSC_1 - TSC_0) * mult >> shift$
end function
```

The general algorithm to compute time using the TSC is shown in algorithm 1. At T_0, the current TSC value is read along with the current time. The time that passed since T_0 can be computed by dividing the difference between the current TSC value and its value at T_0 by the TSC frequency. The time since T_0 can then be added to T_0 to compute the current time. Note that for performance, this computation is always done using integer algebra, and never floating point. Even integer division is costly, therefore most time algorithms use a multiplication and a shift, as shown in algorithm 1. The bigger the multiplication is, the more precise the equivalent division will be. However, if the delta is too big the total value would overflow. Consider a shift of 32 bits, and a TSC at 3GHz. In less than two seconds, the delta between TSC_0 and TSC_1 will reach 4.3G, that is 2^{33}, shifting that value by 32 bits to the left leads to an overflow when using 64bits integer. Before an overflow could happen, the current time is accumulated into the initial values (T_1 becomes T0).

While the OS starts and read T_0 from a hardware Clock and then apply corrections from time to time using NTP, the userlevel application can use the OS clock itself as a first source and apply corrections by re-reading the OS clock from time to time. [76] proposes a userspace high precision time library that automatically selects the best source, however the library is not publicly available and the paper does not provide much implementation details such as when re-synchronization with the time source (the OS wall time system call) is done, and how shifts are applied to catch up for NTP updates that unexpectedly change the source time, if it actually does.

The main problem is that the time in OSes is not advancing at a constant rate as NTP corrections may be applied to slowly catch up with the real time. Therefore we try to compute an accurate time and TSC frequency without any available notion of constant time. A solution would be to base ourself on NTP and not the OS, but that would mean that Click would produce timing and statistics uncorrelated to other systems events, therefore we prefer to synchronize with the OS timing system, no matter how much variable and unprecise it can be, even if that requires more engineering efforts detailed below.

We modified Click to use a function pointer for time computation instead of the hard-coded system call. When Click starts, the OS system calls are always used. The user can then place any Clock element in its configuration that will replace the function pointer when ready to provide an accurate time. This enables to afford waiting to ensure the Clock is stable enough, as the OS can be used in the meantime.

The TSCClock, our element that sets the function pointer to a TSC-based user time facility has 3 phases: stabilization, synchronization, and running.

Stabilization The goal of the stabilization phase is to compute the TSC frequency. Sleeping for one second and computing a delta is not precise enough because the sleep is not guaranteed to be exactly one second, and the time in the OS is not advancing at a constant rate. Therefore loops of $c = rdtsc; sleep(1s); freq = rdtsc - c$ will give variable frequency. In our solution, a timer runs every 10ms and computes the frequency $freq$ that the TSC seemed to have during those last 10ms. It then corrects a base frequency using weighted average, such as $base\_freq = alpha * freq + (1 - alpha) * base\_freq$ until less than an error threshold on the compute time (user-defined, by default 100ns) is achieved for 10 *consecutive* runs. When the precision is not achieved for 10 runs the whole stabilization phase restarts.

Synchronization One problem of the TSC is that it may not be synchronized between CPU cores, though recent CPUs ensure they are. In the synchronization phase, each CPU runs another 10ms timer that compares the OS time and the computed time to ensure that the TSC is synchronized across cores and compute a per-CPU potential TSC offset. When all cores gets good timing for 10 runs, the function pointer is set and the TSCClock enters the running phase. Therefore, **if the TSC is not stable, the synchronization phase will never pass and therefore the userlevel clock will not be activated.**

Running From that point, Click uses the TSCClock to get the current time. The function to compute the time is similar to algorithm 1. In parallel, an "accumulation" timer runs every second to accumulate T_1 in T_0 and reset TSC_0 to the current TSC value for the reasons explained above (mainly, avoiding an integer overflow when computing the time). The accumulation timer also computes the difference between the computed time and the actual OS time to slightly change the TSC frequency to catch up with the OS time. If the catchup is bigger than one second (due to NTP updates), the time jumps directly to the correct time, as the OS does.

As in Click timers are sloppy (they run after all tasks run) one Click task may keep the CPU for so much time that the timer cannot run every second. To prevent that case, if the delta in time is bigger than 2 seconds, the timer moves to another CPU.

If no CPU can ensure a run every two seconds, the TSCClock is uninstalled and the timing function reset to the OS-based one.

The data needed during the running phase to compute the approximated time is the last TSC value (TSC_0), the time when it was read (T_0), and the frequency (kept as a shift and multiplier to avoid 64 bit division). They must be updated atomically together, as if a thread reads the old TSC_0 TSC value but the new T_0 value, the computed time would be wrong. Therefore a *Read-Copy-Update (RCU)* structure is used. RCU will be described in section 4.3.2. In a nutshell, RCU allows very efficient concurrent reader accesses to a structure, while a writer will work on a copy of the structure and commit the modification atomically. This applies very well to the TSC Clock as the data is read when time must be computed, while the data is updated only once every seconds, when the accumulation timer runs.

3.4.2 Evaluation

3.4.2.1 Accuracy

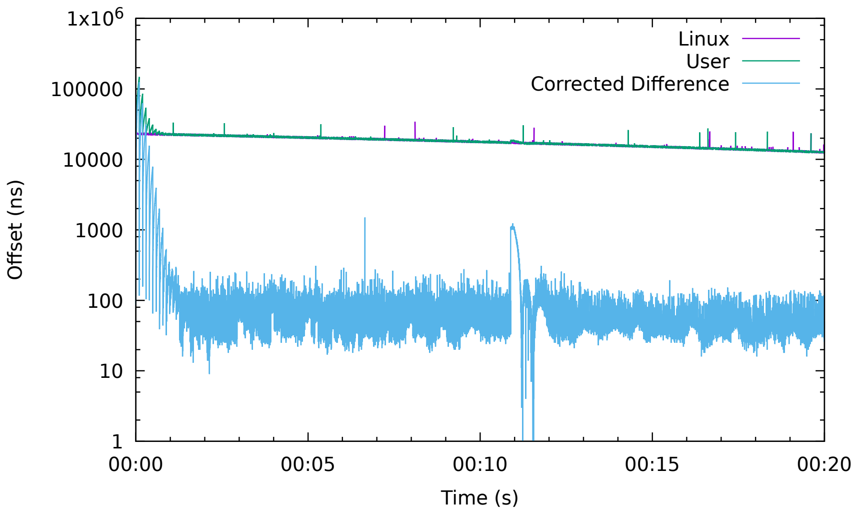


Figure 3.25: User timing accuracy during the 3 phases using the GPS clock as a reference (Linux and User lines), and the difference between the time as read using Linux time and the User time. The difference is corrected by subtracting the time taken for two consecutive user time read to compensate for the delay of the read itself. The stabilization phase starts at 10.89 and the Clock is installed and running at 15.89.

To ensure the accuracy of our system, we compare the Linux time and our User time to a GPS-based high precision PCI-Express clock (Meinberg GPS180PEX). The GPS time is acquired using memory-mapped IO in a very efficient way. Figure 3.25 shows the first seconds of the synchronization phase, computing the difference with the GPS time for both methods and the difference between the two results every $\sim 1ms$. The $\sim 10\mu s$ spikes that can be seen from time to time in the User and Linux lines happen with delta between the GPS clock and both User and Linux time. We believe them to be caused by interrupts that happen during the computation of the time, leading to context switches in certain circumstances and a pause in the middle of the time computation. If the spikes were caused by some inaccuracy of the TSC counter we would observe also "negative" spikes. The corrected difference line in figure 3.25 is the difference between the two times, but subtracted by the time taken to compute the user time itself. To mesure the computation time of the user time itself, we simply compute the time twice, and compare the difference between the results of the two computations. The synchronization phase that starts at 10.89 appears to introduce jitter because it is trying to find a possible local TSC offset for each core which initially amplifies the jitter before converging. The difference is most of the time correct in a +-15ns margin but with a constant offset of 90ns that will be discussed later.

Figure 3.26 shows the accuracy over a longer period of time. We implemented a version of our user clock that sync on the GPS time instead of the OS time, given by the "User - GPS" line in figure 3.26. The offset between the user GPS-based time and the actual GPS time is equal to the time taken between two GPS read (the "GPS read time" line). Similarly, the difference between the "User - Linux" time and the Linux time is equal to the time taken to compute the Linux time itself, *i.e.* the time to call the system call (which is represented by the "User read time" line). In both cases, the user clock actually takes the error of the base Clock. While those constant offsets could be compensated, we follow again the original method which would not compensate for its own time out of the box.

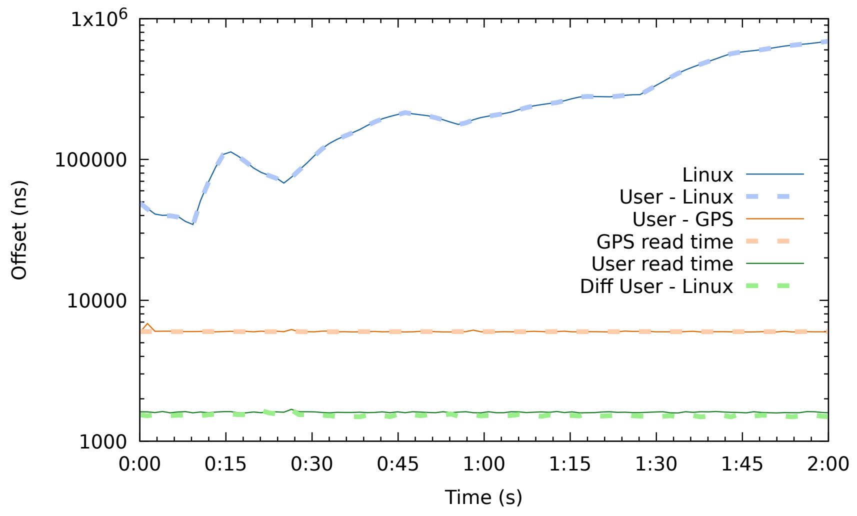


Figure 3.26: Relative accuracy over 2 hours using the GPS clock as reference, lines are approximated for clarity but $\sim 10\mu s$ spikes are still observable on every line as in figure 3.25. User - Linux is the User clock synchronizing with Linux time and User - GPS is following the GPS time. Both read times are time between two consecutive calls to the respectively the GPS device and the user clock (no matter its reference).

3.4.2.2 Performance

Figure 3.24 shows the performance of our userlevel Clock compared with the standard "system call" (a vDSO in practice). Time-stamping all packets of the router test case with the vDSO leads to a 22% impact on throughput. Using our clock instead of the vDSO reduces this impact by 60%.

4

Distributed packet processing

As shown in section 2.3, handling 10 Gbps of minimum-size packets on one core is only possible with a fast framework to quickly deliver packets to userspace and a fast processor. And even in this configuration, any processing must be delegated to another core as the core assigned to the receive loop is nearly submerged.

Therefore, in section 4.1 we review the ways to use multiple CPU cores to process packets from a single input device.

In section 4.2 we study techniques to seamlessly handle multi-threading and multi-queuing in FastClick, making the configuration simple by automatically detecting thread traversal in network functions. FastClick uses graph analysis to discover the path that each thread can take and minimizes the use of memory and multi-queue.

Finally, in section 4.3 we review data-structures that enable very efficient parallel processing with no or a minimal amount of locking.

In this section, DPDK will be used in experiments as chapter 3 showed that it is more or less on a par with Netmap and we don't need to re-experiment with both I/O frameworks.

This chapter in a nutshell

▶ Context of this chapter

- There are two main approaches for distributed packet processing with multiple cores: pipelining and parallel processing.

- While there are studies comparing the two approaches, the limits of the approaches with more than two cores and when they should be used in the context of userlevel processing, batching, and with modern hardware is still not fully reviewed.
- Operating Systems locking facilities are too slow in the context of high-speed packet processing and do not cope well with a run-to-completion model based on userlevel threads, which is used in Click and most recent NFV platforms.

▶ **Highlight of our main contributions in this chapter**

- We show the parallel approach should be preferred in most cases after a very extensive study under various kind of workloads. This is because it avoids atomic operations and data cache misses that occur when passing packets from core to core as needed by the pipeline approach. Even if instruction cache may suffer from the parallel approach, cases are very rare where the pipeline approach actually performs better, in part because compute batching amortizes over a batch the cost of warming up the instruction cache.
- FastClick detects thread traversal and enables a novel easy-to-use multi-threaded and safe configuration, hiding multi-threading and multi-queuing specifics.
- We review and propose network-specific datastructures for parallelization according to their potential usages. We propose a novel userlevel RCU mechanism for efficient read-mostly applications, and RxWMP, an exclusive multiple-readers or multiple-writers facility around a per-thread duplicated structure that does not degenerate in either read-mostly or write-mostly.

■

4.1 Distribution approaches

When a lot of work has to be done on multiple packets, there are multiple ways to take advantage of multiple CPU cores to accelerate the processing.

A first solution is to use multi-queueing to implement a "parallel" approach, not only to avoid locking in the output hardware queue for the full push mode as proposed in section 3.2.5, but to exploit functionality such as *Receive Side Scaling (RSS)* which partitions the received packets among multiple hardware queues. Each hardware queue can then be serviced by different cores. This is depicted in figure 4.1 (A). In the parallel approach, the *data* (the packets) is distributed among the cores, that all executes the full processing in parallel.

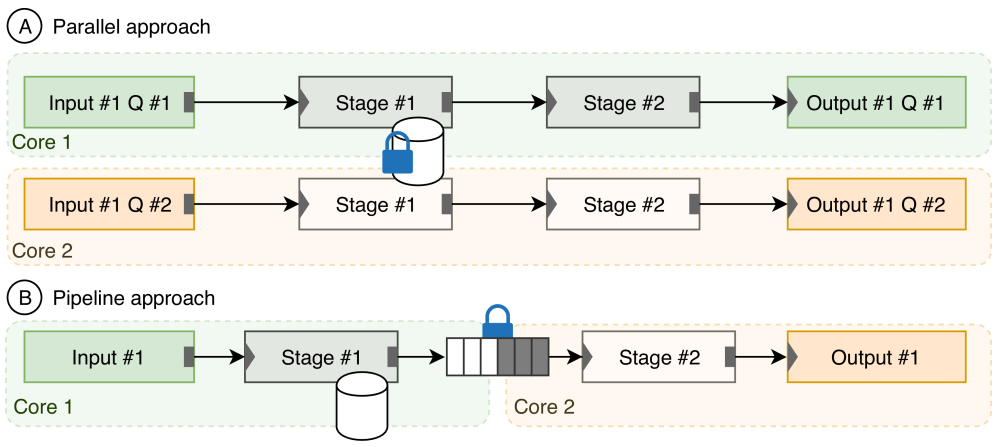


Figure 4.1: Distribution of two processing stages among two cores using the parallel approach with hardware multi-queues and the pipeline approach

A second solution is the "pipeline" approach, that distributes the *instructions* among the cores. Each core does some part of the processing and then passes the packets to the next core. In this chapter, we'll refer to "part of the processing" as *processing stages*. Together, the chain of processing stages represents all the work to be executed on every packets. This solution is presented in depicted 4.1 (B). While we only show figures where the pipeline approach always executes one processing stage, we actually vary the amount of work per processing stages, simulating the case where each core runs multiple processing stage.

Comparison regarding CPU cache locality. In Click, and generally for efficient implementation of the parallel approach, the code is not duplicated. Only some mutable state for each element is duplicated. Therefore in both approaches the total number of instructions is identical. However, in the parallel approach, the instructions will have to be fetched in every L1i caches. But the parallel approach improves data cache locality as the packet always stays on the same core and can be kept in local L1d or L2 cache, while the pipeline approach improves instructions cache locality as each core executes a reduced set of instructions that can better fit in L1i or L2. The pipeline approach also benefits memory that is used specifically by one processing stage such as DPI rules or firewall rules that will benefit from staying in a cache as close as possible to the core using the rules.

Comparison regarding locking. The first approach (parallel) needs locks and complex data structures when the state must be synchronized between all cores, *e.g.* for flow tables. While the second approach (pipeline) introduce some of the same problems as the push-to-pull path discussed in section 3.2.3 to enable multiple cores to exchange packets and notify the next core that some packets are available. However pipelining (the second model) leads to easier state management as the same function is most of the time executed by one unique CPU core. Even when pipelining is involved, some status such as NAT flow tables may still need to be shared between cores handling different side of the connection, or status of connections shared between the cores handling the first part and the last part of the pipeline.

[16] already compared the two approaches for using 2 cores to accelerate processing. They evaluated the impact of doing N memory accesses to an array of size S for each packet, varying both parameters to see the impact in term of throughput. We re-built the experiment to study the impact also in term of latency, and for multiple other reasons.

First, the push-to-pull path in itself is responsible for some performance loss because the select mechanism is heavy, and the push-to-pull path leads to constantly scheduling and de-scheduling tasks. Section 3.2.3 studied the impact of push-to-pull path in more details.

Secondly, the Click Queue implementation is also perfectible when the number of cores goes up. Therefore in the following experiment, we keep a full-push path using the new Pipeliner element described in section 3.2.3.1 that will benefit the pipelining approach.

Thirdly, with compute batching elements process packets in batches. That means that even if the instructions are not in the closest CPU cache for the first packet, they will be for all the subsequent packets of the batch. This will mostly benefit the parallel case, but also the pipeline mode when the amount of instructions per core is still too big for the L1i cache. Also, [16] used kernel click and some mechanisms are heavier in userlevel (like adding and removing file descriptors for the select operation) or, at the contrary lighter such as the packet reception cost when using I/O frameworks like DPDK or Netmap. Finally, pipelining may suffer from more synchronization work when the number of cores, and therefore the number of software queues is going up and is not modelized in [16].

4.1.1 CPU-bound workload comparison

The two main drawbacks of the pipeline approach are the software queues that introduce synchronization cost between the multiple cores handling the software queue and the fact that packets will be allocated on one core but released on another one. The synchronization cost can be amortized with batching. We'll use the software queues to pass batches of packets instead of single packets, amortizing the synchronization costs per-batch. The second drawback is that Click packets are allocated on the first core

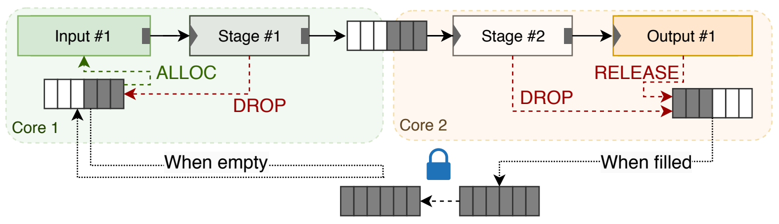


Figure 4.2: Pool allocation and release process under the pipeline approach

of the pipeline but are recycled on the last core as shown in figure 4.2. In Click, the packet pool have a per-core LIFO cache. When the LIFO cache is reaching a default size of 2048 packets, it is considered full and transferred to a lock-protected global pool as a batch of 2048 packets. In pipeline mode, it is the last core that will release packets after transmission and fill the cache. The first core of the pipeline will access the global pool to recover the batch and re-fill its CPU local cache when its local pool is empty. Therefore a larger number of packet descriptors will be accessed in the process, while the parallel approach can re-use the same packet descriptors in the LIFO queue, keeping most probably all packet descriptors in cache. To compensate for this drawback, we

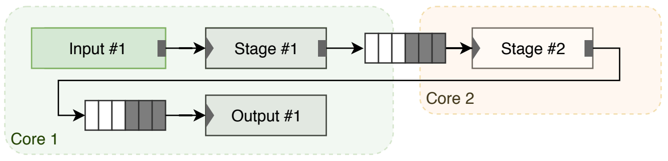


Figure 4.3: Pipeline approach to distribute two processing stages among two cores using one more software queue to transmit packets using the same core than the one receiving them

propose a "returning pipeline" approach that introduces one more software queue to send packets on the core where they were originally received. Therefore the recycling

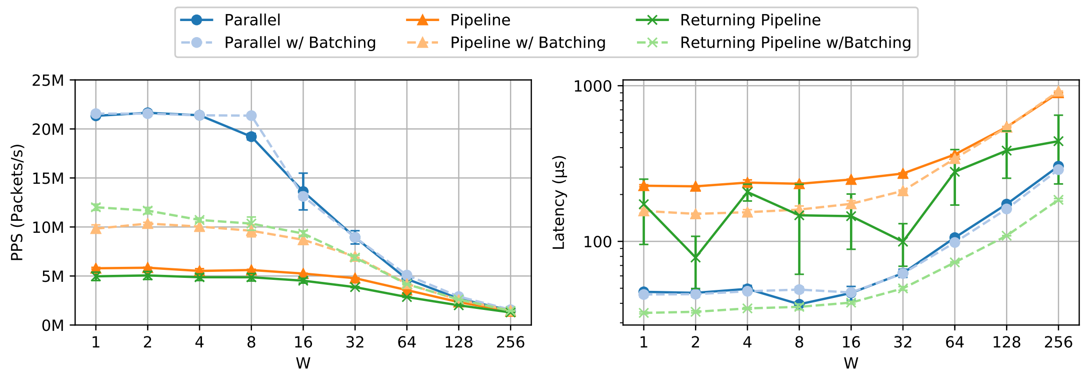


Figure 4.4: Performance of the parallel and pipeline approaches to execute two processing stages under increasing CPU workload. Note that 22MPPS is actually the limit of the generator. Latency is log scale.

and the allocation of packets are done on the same core as shown in figure 4.3. The drawback of the returning pipeline approach is that the first core will have more work to do than the second has the first core will handle one processing stage plus the full I/O routines. Running the I/O operations on a dedicated core that does not execute any processing stage will be considered later.

We built a test case using two cores to run two processing stages as in figure 4.1, but without shared state between processing stages as suggested on the figure. Each of the two processing stages are generating W pseudo-random numbers for each packet passing by. We use the standard *std::mt19937* C++ generator, which is a CPU-intensive operation. The last core handling the packet is responsible for rewriting the packet MAC addresses before transmission. The testbed is the same as in section 3.2.1 for the 40G campus router test case but we generate 64bytes UDP packets instead of replaying traces.

Figure 4.4 shows a comparison of the parallel approach, the simple pipeline approach and the returning pipeline approach. All of them with and without batching. The parallel approach is performing better than the pipeline one in term of throughput. Batching enables amortizing the synchronization cost, but it still leaves the pipeline approach around 1.4 to 2 times slower than the parallel one. It is interesting to see that the returning pipeline approach performs better than the simple pipeline approach with batching but worst in term of throughput without batching. This is because the returning cost is much higher as the pipeliner is passing back packets one at a time to the first core for transmission instead of doing it per-batch.

In term of latency the returning pipeline approach with batching seems to perform slightly better than the parallel approach. It is because in the parallel approach we use one input hardware queue per core, therefore when the rate is higher than what the cores can handle (because the input rate is higher than the processing rate), the input hardware queue are beginning to fill up. In the experiment, as we use two cores, we buffer twice more packets as the total amount of buffers in the hardware queues is doubled. Therefore the average latency increases.

Given the results, all the following use batching and it will not be explicitly mentioned any-more.

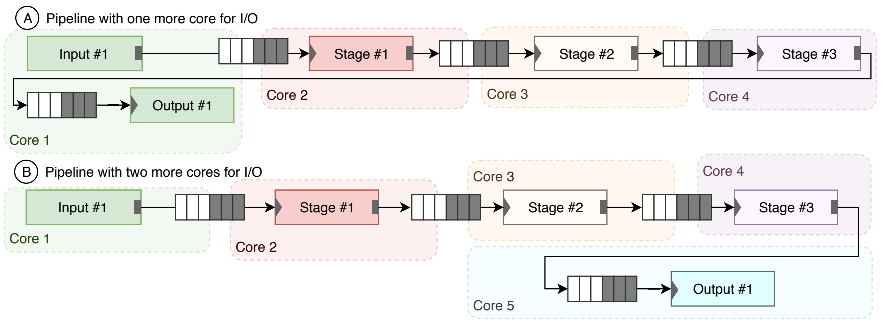


Figure 4.5: Two new pipeline approaches to distribute 3 processing stages among 3 cores, using one or two more cores to receive and transmit packets

All approaches tend to lead to the same performance when the amount of work per packet (W) is huge, as the share of CPU time for the packet transfer is smaller and is therefore only a small part of the overall work to be done. Letting the CPU suffocate under so much constraint is not very realistic, instead the work to be done can be distributed among more than two cores.

Figure 4.5 shows two pipeline configuration using 3 cores to run 3 processing stages, using one or two more cores to handle I/O. The idea behind those configurations is that when the number of cores increases in the pipeline, it may be better to use the core budget to run the I/O elements on their own cores. Indeed, the core(s) running the I/O elements in the pipeline or returning pipeline approaches also run a processing stage and therefore may become bottlenecks.

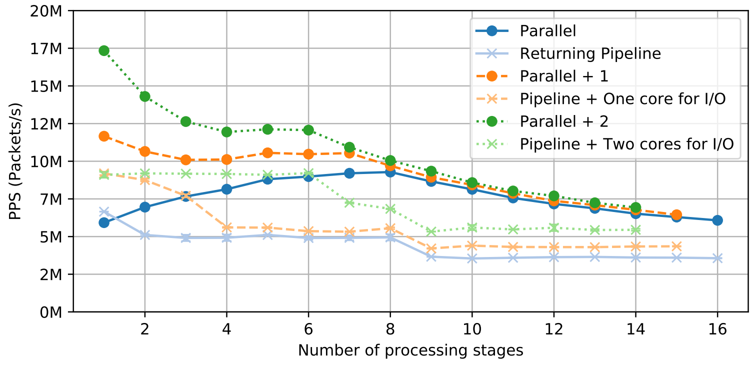


Figure 4.6: Throughput of the parallel and pipeline approaches with a varying number of processing stages. Parallel and Returning pipeline use as many logical cores as processing stages. Parallel + 1 and Pipeline + One dedicated core for I/O use one more core than the number of processing stages, while the last two use two more logical cores.

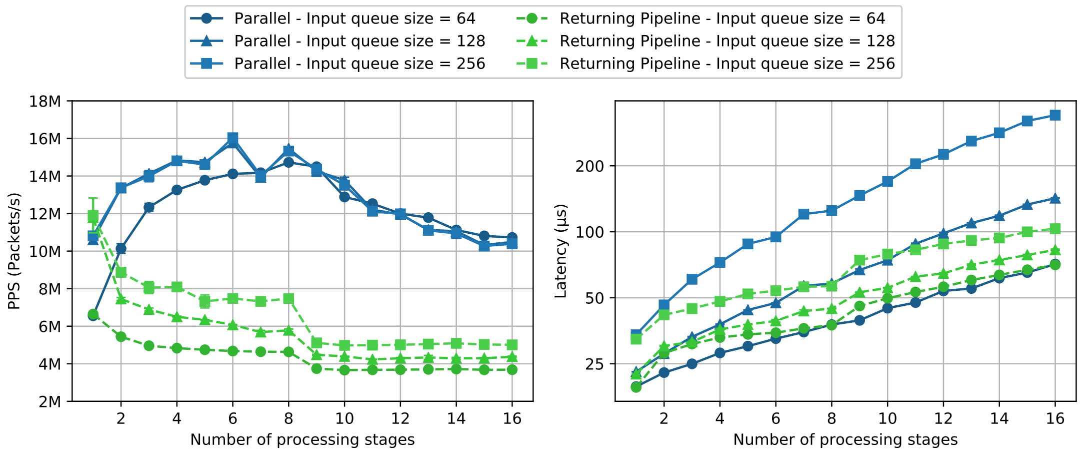


Figure 4.7: Performance of the parallel approach and the returning pipeline approach with a varying number of processing stages. Both approaches use as many logical cores as processing stages. Each method is compared using 64, 128, or 256 packets descriptors in the hardware input queue of the NIC used

Our device under test has an Intel$^{®}$ Xeon$^{®}$ CPU E5-2630 v3 @ 2.40GHz which has 8 physical cores, but 16 logical cores with hyper threading enabled. Therefore in the following, figures using more than 8 logical cores spread on shared physical cores. Logical cores 9 to 16 are hyper-threads corresponding to cores 1 to 8. Figure 4.6 shows the performance of those two new pipeline variants with W still fixed to 32. In this new test, the number of processing stages now ranges from one to sixteen. The parallel and returning pipeline approaches use as many logical cores as processing stages. The two other approaches using dedicated I/O logical cores use one or two more logical cores for I/O. To allow a fair comparison with the parallel approach we also re-ran the parallel approach with one or two more logical cores than the number of processing stages to compare fairly against the dedicated approaches using an equivalent number of logical core. Therefore in figure 4.6 the methods are to be compared two by two (groups using the same final number of cores have the same line type and colours but different points for different methods).

The figure clearly shows that while using dedicated I/O logical cores improves the pipeline approach itself, the parallel approach can use more logical cores much more efficiently. The pipeline approach with dedicated input and output cores is very stable because it is actually bottlenecked by the speed of a single CPU executing its processing stages.

Figure 4.7 shows the packet processing rate and the latency of the parallel and returning pipeline approaches with various input hardware queue sizes. As the number of logical cores increases, we use more input hardware queues in the parallel approach and actually buffer much more packets and add up latency, as previously established. The figure shows that the latency can be decreased by using fewer descriptors in the input queues. However, the total buffering capacity should be taken in consideration to avoid dropping too much packets. In realistic situations, lowering too much the number of descriptors would lead to TCP retransmissions that will put even more pressure on the link. The outcome of this experiment is not that using less descriptors leads to better latency, but that at equivalent buffering capacity the parallel approach performs better, while in all considered cases up to this point, achieving better throughput too.

The pipeline approach using software queues could start to buffer as well. It does not happen in these micro-benchmarks because the pipeline stages are well balanced and software queues stay empty most of the time. In a more realistic situation, the pipeline approach would also start buffering more packets when the number of logical cores increases.

4.1.2 Memory-bound workload comparison

As the pipeline approach is supposed to perform better in term of memory locality, we now compare the pipeline and parallel approaches under a various number of memory access to a various amount of data.

As in [16] we modify the processing stages to do N memory access for each packet to an array of size S Mbytes. The array is filled with random data. The size S is per-processing stage, therefore when using 2 processing stages the total memory pressure on the shared L3 cache of the CPU is $2 * S$. For each packet, the processing stage generates one random number using the same method than in 4.1.1 ($W = 1$) and accesses a random element of the array using that generated number. As shown in the previous section, generating a new pseudo-random number for each N access of each packet would kill the performances quite quickly. Instead, the random number is mixed (using a xor operation) with the content of the array we just read, and used as the index of the next element to read in the array. This is then repeated N times to execute N access to the array.

As can be seen in figure 4.8, the parallel approach performs better when the number of accesses is low ($N < 20$) or high($N > 1000$), especially when the size of the data is small ($S < 8$) or high ($S > 64$). When the returning pipeline approach performs slightly better, it is per a few percents while the parallel approach can lead up to a 2X improvement. The second graph of figure 4.8 zooms on the range best for pipeline approach. The advantage of the pipeline approach for the array locality allows the returning pipeline approach to relatively catch up with the performance loss reviewed in the CPU-bound workload comparison.

We also add a parameter R which is the percentage of the N access made to the content of the packet instead of accessing the array. Therefore if R is 0%, we only read from the memory array and if R is 100%, we only read from the packet.

Figure 4.9 repeats the experiment, showing the improvement of the parallel approach over the returning pipeline approach but this time with an increasing proportion (R) of the memory access made to the packet content instead of the array. The number

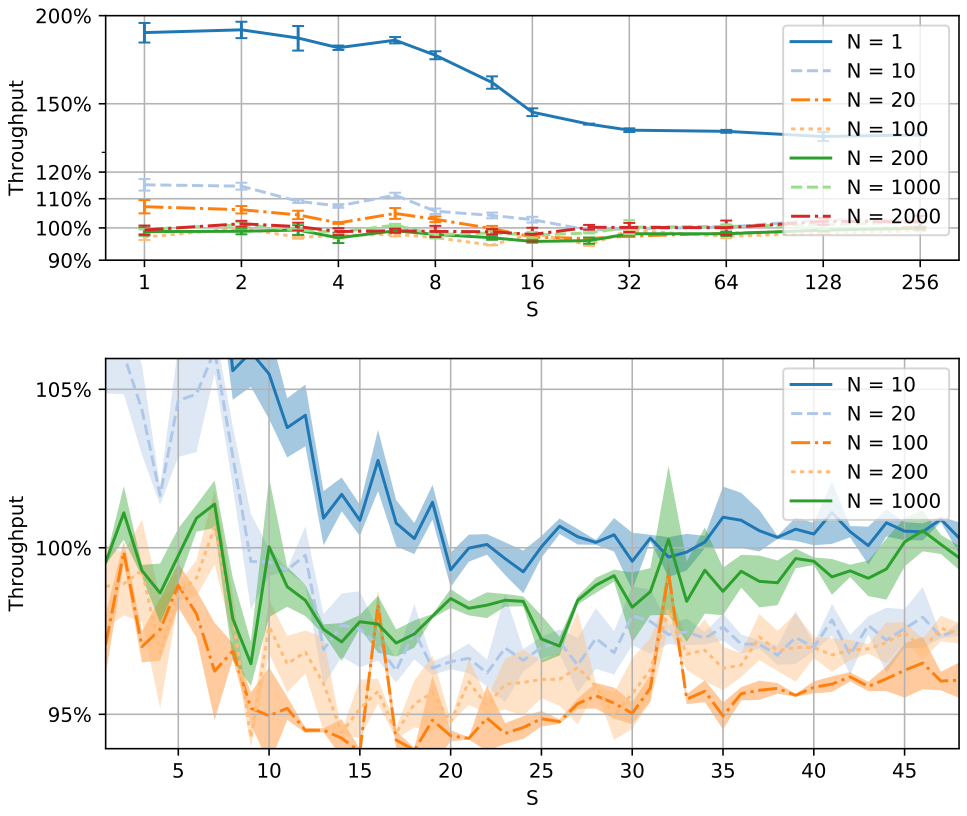


Figure 4.8: Relative performance of the parallel approach over the returning pipeline approach using two processing stages running on two cores. The CPU shared L3 cache is 20M. Local L2 cache is 2M. L1 caches are 256K.

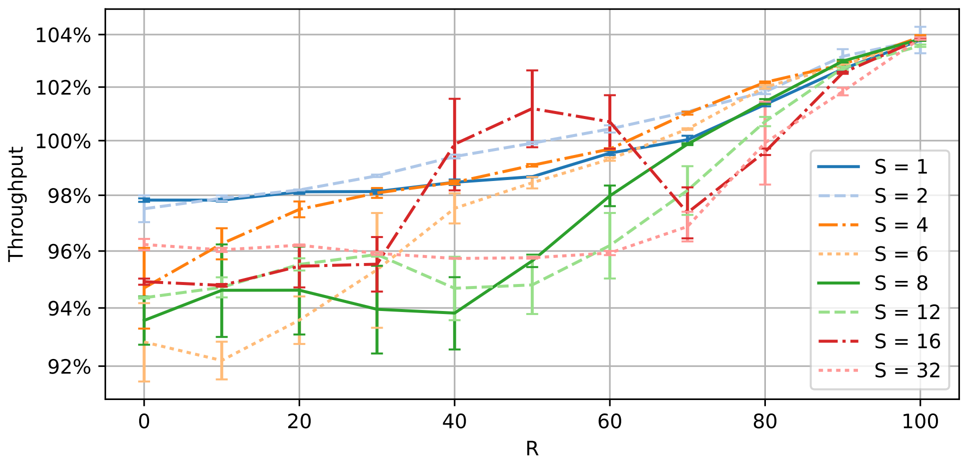


Figure 4.9: Relative improvement of the parallel approach over the returning pipeline approach using four processing stages running on four cores under an increasing percentage of access to the packet data instead of the array (R).

of access N is fixed to 100, which showed to be the most advantageous value for the pipeline approach.

As R increase, the parallel approach performs slightly better as the packets may stay in a lower cache level along the path, while with the pipeline approach each access to the packet on a new core needs to bring the packet in a closer cache. The difference between the two approaches is never very high because in most cases the data is already available in L3.

The higher R range where the parallel approach is performing best is in fact the case of most network functions. Most functions do a simple processing (very low S, low N) according to one or two fields of the packet as opposed to some static data (high R). We used 4 cores to run 4 processing stages in this test, but figures in appendices A show the trend is identical for 2 and 8 processing stages/cores. The parallel approach still keeps its advantage with a low ($N = 20$) number of memory accesses per packet. With a high number of accesses ($N = 2000$), as the packet is much smaller than the array size, making 2000 access to a very small amount of memory makes both approaches behave as under a purely CPU bound task. Still, the parallel approach behaves mostly better.

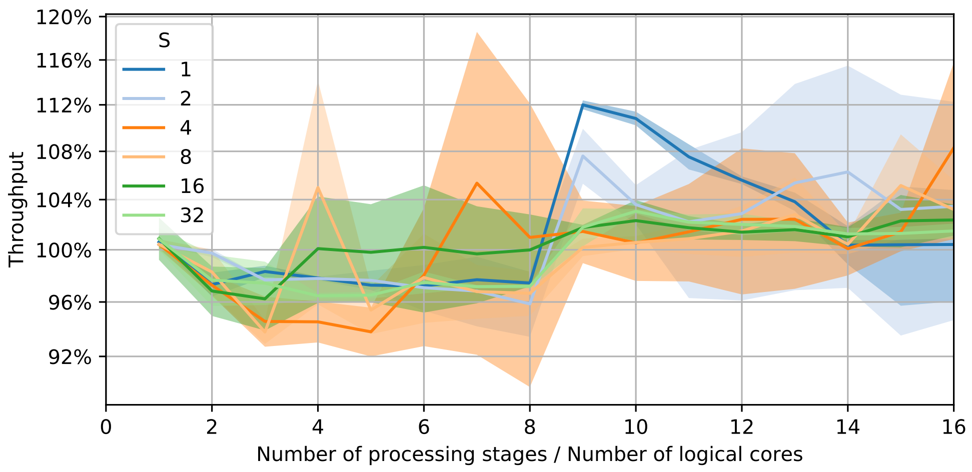


Figure 4.10: Relative performance of the parallel approach over the returning pipeline approach doing $N = 100$ memory access to an array of size S for an increasing number of processing stages. One logical core is used per processing stages.

Finally, the number of cores involved may influence performance under a memory bound situation, as it did for the CPU-bound situation where the parallel approach performed best with more cores. Figure 4.10 shows the relative performance of the parallel approach over the returning pipeline approach under an increasing number of cores for $N = 100$ access to the array ($R = 0$), among the most advantageous values for the pipeline approach. The pipeline approach is around 2 to 5% more effective when using 2 to 8 cores. However the parallel approach is able to use much more efficiently hyper-threading and performs up to 10% better than the pipeline approach when using the hyper-threads. Appendix A shows the same figure for $N = 20$ memory access per packet in A.5 and $N = 2000$ in A.6. With a low number of accesses, results are identical except for small array size where the parallel approach can keep the array in L2 and therefore performs better. With a higher number of accesses, the test behave in a CPU-bound fashion and the parallel approach performs much better as previously shown.

This memory-bound performance study showed that the situation where the pipeline approaches perform better than the parallel approach is in fact quite limited. The parallel approach is performing better when the number of access to some static data is either small ($N < 20$) because the memory pattern does not influence much the results, or high ($N > 2000$) leading to a situation similar to the CPU-bound behaviour. However, even between those constraints if the proportion of access to the packet instead of the static data is high, or if hyper-threads are used the parallel approach performs better again. Finally, we could achieve up to around 10% better performances when all those conditions were reunited but the parallel approach can perform up to 100% better in some other but often seen conditions. That is a few access ($N = 1 \sim 5$) to a few amount of static data ($S < 1$) and a small amount of fields ($R \simeq 50$) which is a common case in VNFs.

4.1.3 Work distribution bias in the pipeline and parallel approaches

Both approaches have an imbalance problem. With the parallel approach, *Receive Side Scaling (RSS)* is used to hash the packets and distribute them among cores. Nothing guarantees that cores receive the same number of packets. With the pipeline approach, it may be hard to fairly split the workload into processing stages of exactly equal shares.

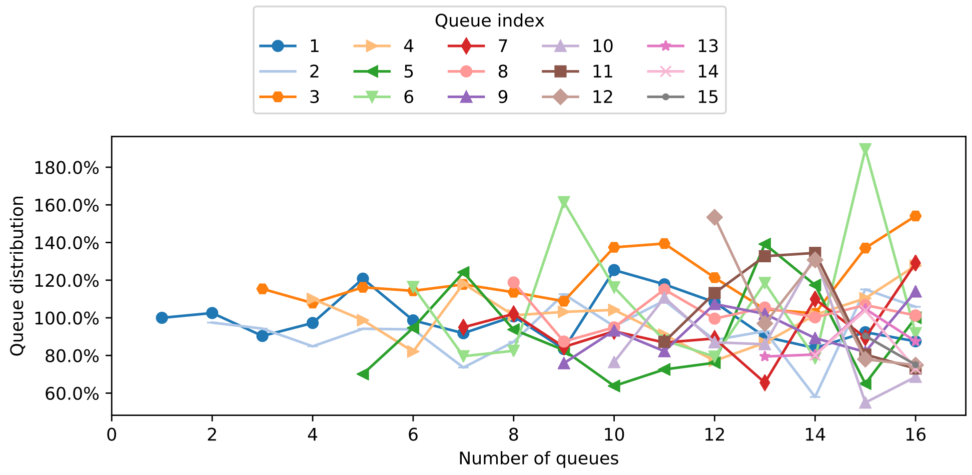


Figure 4.11: Deviation from a uniform distribution of packets among hardware queues using RSS hashing with a varying number of hardware queues.

Figure 4.11 shows the number of packets each hardware queue receives using our campus traces. When using 15 queues, the queue number 6 receive $\sim$ 190% of the amount of traffic a perfect distribution would achieve. While the queue 10 receives around $\sim$ 50% of it. It means that some flows will have much more latency than others, maybe even dropping packets. The CPU handling the queue 6 will have 4 times more packets to handle than the CPU handling queue 10. To circumvent the problem, RSS

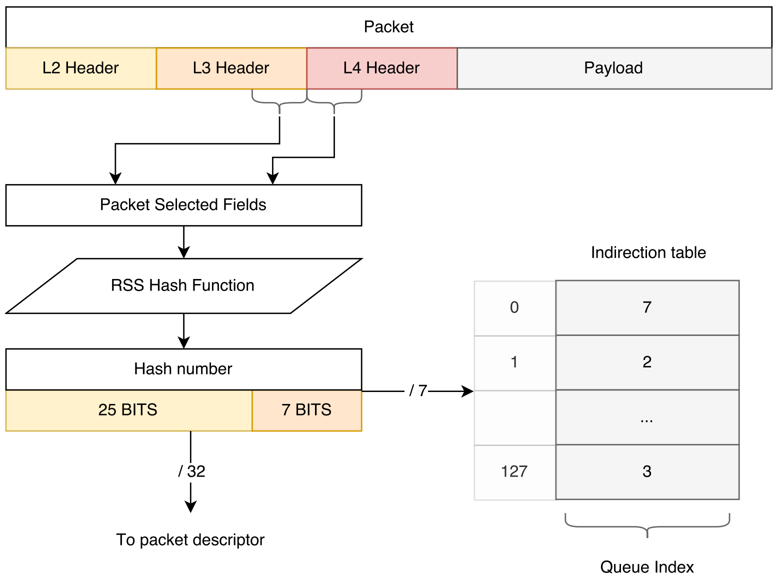


Figure 4.12: RSS inner working. Selected packet fields are hashed to a 32 bit number. The 7 least significant bits are used as an index inside an indirection table that gives the queue index

uses an indirection table to match the 7 last bits of the hash to a queue index, allowing to re-balance the load if need be as shown in figure 4.12. However when doing stateful processing changing the indirection table would mean that some flows may be served by a different core than the one it was currently being served on. In which case some synchronization would be needed such as using a protected global flow table instead of a per-CPU one, or support migration of existing flows between cores. Therefore this functionality is actually rarely used in practice. In fact, it was not even supported by the ixgbe driver[1] in the Linux Kernel before we upstreamed a patch ourselves into the mainline Linux Kernel as part of an unrelated project [77].

With pipelining, the imbalance problem is that the number of processing stages does not always fit the number of cores. The work to be done is not necessarily splittable in small chunks, such as with IPsec encryption or when the load cannot be known in advance such as with DPI where the processing time will depend on the rules and if they match quickly the payload or not.

[1]ixgbe is the driver supporting the 10G Intel 82599 chipset often seen in many research papers

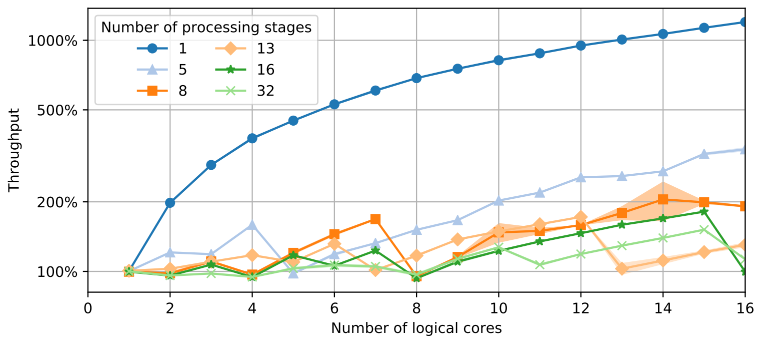


Figure 4.13: Relative performance of the parallel approaches over the returning pipeline approach to run a given number of processing stages using an increasing number of cores. Other parameters are chosen among the most advantageous for the pipeline approach (N=100, R=0, S=2)

As a last test, we tried to distribute some fixed number of processing stages using an increasing number of CPU cores. Other parameters are chosen among the most advantageous for the pipeline approach ($N = 100$, $R = 0$, $S = 2$). For this test in pipeline mode, cores are assigned in contiguous order as best as possible to distribute processing stages among cores. When the number of logical cores perfectly divides the number of processing stages, the pipeline approach performs slightly better given the chosen fixed parameters as shown in figure 4.13. However in other cases some cores are doing more work than others and become bottlenecks.

We did not find a case where the pipeline approach did perform much better than the parallel approach, but it does not mean that there are none. There may be certain cases where a hybrid approach may lead to better performances. But we showed that while the common belief that pipelining allows to improve CPU caches efficiency is not false, there are in fact very few cases where it actually leads to enough improvement to compensate for the inherent slowness of synchronizing and passing data between cores. It is also partly because compute batching allows to amortize the relative worst locality of the parallel approach.

4.2 Handling mutable data

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 data caches, statistics, counters, etc. In figure 4.14, the little dots in Routing elements represent per-thread duplicable meta-data, like the cache of a last seen IP route, and the black dots are 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 directions of the flow, etc).

In vanilla Click in a multi-queue configuration such as proposed by RouteBricks[15], there will be one FromDevice element attached to one hardware input queue of each device. Each cores handles one queue from each device, as shown in figure 4.14 (a). The problem is that in most cases, the Click paths cross at one element that could have mutable data.

A first approach is to use thread-safe elements on the part of the path that multiple threads can follow as in figure 4.14 (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)* that contains the forwarding rules of a routing element. This method is advantageous in cases where memory duplication is too costly, *e.g.* in the case of a big FIB, although the corresponding data structure must become thread safe if it may be modified. Moreover, the operator must use a special element to either separate the path per-thread (as shown by the white circle in 4.14 (a)) to use one output hardware queue for each thread as for the input, or use a software thread-safe queue before reaching the output hardware 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 4.14 (b).

A third approach would be to duplicate the element for each thread path with a shared pointer to a common non-mutable data like in figure 4.14 (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).

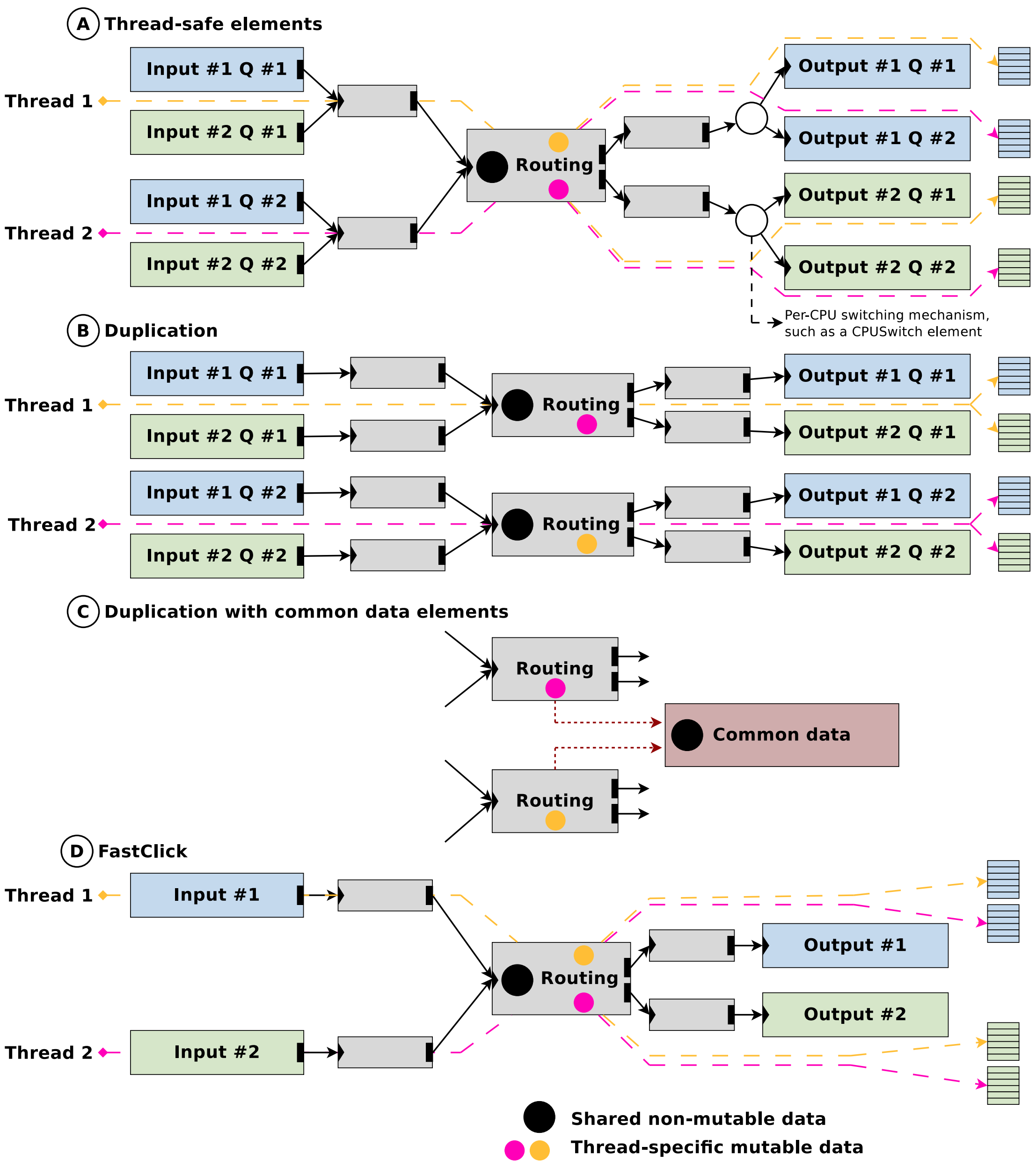


Figure 4.14: Three ways to handle data contention problem with multi-queue and our solution.

We prefer to go back to the vanilla Click model of having one Element representing one input device and one representing the output device. In that way the user only cares about the logical path and the queueing management is hidden.

FastClick supports two thread allocation schemes. The first one allocates one hardware queue per core for each devices, achieving the same configuration than in figure 4.14 (a). We refer to this as the “balanced” mode, as one core will serve packets from one hardware queue of each devices. The second ones exclusively allocates cores to input devices, which is depicted in figure 4.14 (d). We refer to this as the “dedicated cores” mode as each input device has one or more cores exclusively dedicated to serve its packets. Which mode to use depends mostly on the use case. Having each core handling one queue of each device enables load-balancing if some input devices 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. This is more common in NFV schemes, where network functions, and sometimes even full virtual machines have dedicated cores.

Our FastClick implementation takes care of allocating queues and threads in a NUMA-aware way by creating a Click task for each core allocated to an input device, without multiplicating elements. On Intel platforms, since the Nehalem architecture introduced in 2008, the Northbridge has been integrated in the CPU. This means that the memory controller, but also PCIe lines are directly connected to the CPU (sometimes through a PCIe switch). Therefore, on recent multi-processor systems, an access to a NIC attached to a first CPU from another CPU will be done through the QPI link interconnecting the CPUs, introducing a performance and latency hit. Therefore on this architecture, serving NICs input queues with the right CPU cores is of importance.

FastClick transparently manage the multi-queueing so the operator does not need to separate paths according to threads or join all threads using a software queue or a CPU-switch element as in figure 4.14 (a).

Assignation of cores to input devices In both modes, we use only one thread per core as we use a run-to-completion model. However, in Click, multiple tasks can be assigned to the same thread (and therefore, to the same core).

In “balanced” mode, for each input device we simply spawn one task per thread running on the same NUMA node than the device. Therefore each core will run one thread, that executes as many task as there are input devices attached to the same NUMA node. With DPDK, we use one hardware queue per task, 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.

To assign the cores to the input device in “dedicated cores” mode we do as follows: 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.

In both mode, the FromDevice element has an argument to disable NUMA awareness and ignore the socket assignation of the device. In “balanced” mode each input device will be served by all cores independently of their NUMA node, while in dedicated cores all cores will be distributed to input devices independently of their NUMA node.

Assignation of queues to output devices 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 behaviour for an

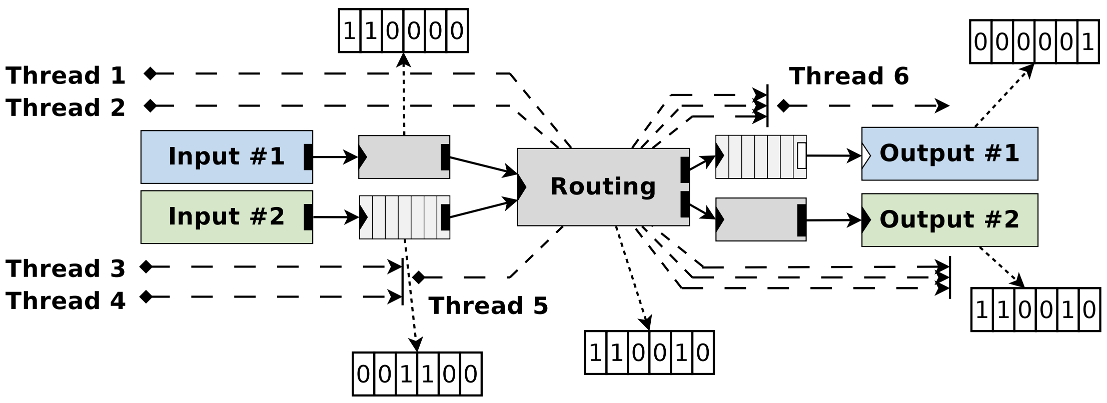


Figure 4.15: Thread bit vectors used to know which thread can pass through which elements.

element without specific getThreads() function. An example is shown in figure 4.15. In that example, some path contains a software queue where multiple threads will push packets. As only one thread takes packets 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 hardware queues are available, the thread vector allows to automatically find if the output element needs to share one queue among some threads and therefore needs to lock before accessing the output ring.

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 software 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 3.2.3.

Our model has the advantages of figure 4.14 (a) and (c) while hiding the multi-queue and thread management and the simplicity of approach (b).

Implementation details: *the per_thread<T> structure*

We also provide a per_thread template using the thread vector to duplicate any structure per-thread, using C++ facilities to easily access the current thread's bucket with the dereference operator ($->$). per_thread makes the implementation of thread-safe elements easier. This is important as the chosen model needs to make the internal mutable state of each element thread-safe, requiring a modification to most stateful elements that therefore needs to be as easy as possible.

Many FastClick thread-safe elements use the per_thread template to duplicate the object T per-thread As the CPU caches work with cache lines (64bytes on our x86_64 CPUs), if two variables in the same cache line are accessed by multiple CPU cores, they would conflict and be invalidated even if the underlying memory is well segmented per-core, a problem known as *false-sharing.* The object T is padded with enough bytes to protect against false-sharing.

At initialization a vector is initialized with as many threads replication of the padded T structure. At runtime, the good bucket is accessed using a global Click thread-local variable stored using *Thread Local Storage (TLS)* that gives the thread's index and use it as index of the vector. TLS themselves cannot be dynamically instantiated, therefore per_thread itself cannot use TLS to implement the per-thread bucket.

■

4.2.1 Ensuring graph thread-safeness

In Click, elements that are multithread-safe are marked with a macro at the end of the file, but this is only used for documentation purposes. Click lexer has been modified to expose the information that the element is to be considered thread-safe as a flag of the element, accessible at run-time. To ensure elements of the graph that are not multithread-safe are not traversed by multiple threads, we add a verification at the router instantiation. We ensure that elements traversed by multiple threads as shown by the thread vector are indeed thread-safe.

The modular approach of Click leads to most elements being quite compact and relatively simple. Therefore it is not a complex task to review elements and mark them as thread safe. There are mainly 2 aspects to verify:

- Element state: if the per-element data is modified by concurrent threads, suitable protections and locks are needed. We review usual solutions in section 4.3.

- Handlers: Click use handlers, similar to functions exposed by elements to enable the access to the element state and modify its configuration. Handlers run in the context of the calling thread, therefore it may lead to multiple threads accessing the same data. As most handlers affect some configuration that is only accessed read-only per the element itself, protecting the state with *Read-Copy-Update (RCU)* data structures is most often the best solution. RCU is a data structure that enables efficient access in a read mostly situation. RCU will be discussed in more details in section 4.3.

4.3 Networking data structures for parallelization

Kernel provided locking and exclusion facilities such as mutexes rely on context-switches to allow other threads to execute while the protected data is locked. In the Click Modular Router, all tasks of the current thread would be stalled if the thread was to sleep. Most high-speed I/O frameworks are built with the aim that each thread is affinitized to one core, and in fact DPDK does not even support multiple threads per CPUs. Therefore specific synchronization data-structures (or spin-looping as last resort, but never for too long) must be used instead of relying on Kernel facilities. The userlevel thread model is followed by most high-speed frameworks because context switches and system calls are too costly as discussed in chapter 2. Therefore in the following we assume that each thread is exclusively pinned to one core.

Some network functions can run in parallel, either by duplicating the mutable state, by using only atomic instructions to access it, by locking to protect concurrent readers, or use specific data structures such as *Read-Copy-Update (RCU)* that will be explained further.

Which one to use entirely depends on the characterization of the network function and the allowed margin for inconsistency. *I.e.* if concurrent reads and writes can happen or if they are to be strictly exclusive. The first axis of characterization is the amount of read versus write. Some locking facilities perform better when accessed for reading mostly or at the contrary when accessed for writing mostly. The second axis is the atomicity needed for the data. One may accept stale data, which could be slightly old or allow for inconsistent data, that is reading a structure partially updated. The following sections study in details what are the best options according to both axes and potential use cases.

4.3.1 Write mostly

In this section, we'll review some locking facilities in a write mostly context. We'll start from the most lax solutions regarding constraint on the consistency of the data structure when multiple writers access it concurrently towards the most restrictive.

Inconsistent Statistic functions such as packet counters can use one counting variable per thread, and compute the overall sum when the count needs to be re-conciliated, which should not happen too often.

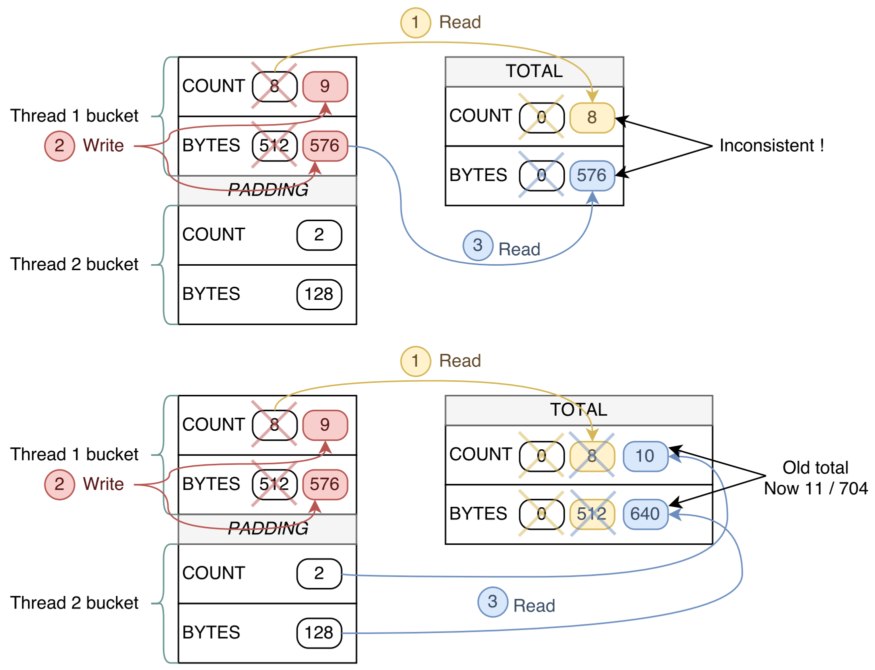


Figure 4.16: Per-thread duplication approach. Two sequences of events leading to an inconsistent read and a stall read respectively

However, as is, the per-thread "duplication" solution does not provide consistency or protect against stale data. Figure 4.16 shows an order of event that leads to the two situations with a packet counter example. It counts the number of packets and the total number of bytes passing by one path of the Click graph. For simplicity and demonstration purposes, we assume that all packets are of 64 bytes. If the per-thread memory is bigger than the architecture word, both values cannot be read atomically. If the reader reads the first value, but the second value is updated before it is read, then the reader will have an unsynchronized copy of the bucket. In the example of figure 4.16, the first reader reads the number of packets that is 8. Then the writer updates the number of packets and bytes to 9 and 576. The reader then reads the number of bytes that is 576. The given average size of a packet would be given as 576/8 = 72, instead of 64. While one thread is summing all per-thread counters, individual threads continue to update their values. By the time the aggregation thread reaches the last thread bucket, the first one may have changed. In the bottom part of figure 4.16, the sequence of action leads to a count of 10 packets for a total of 640 bytes, while at the very same time the count is actually 11 packets for a total of 704 bytes. It is possible for the two situations to happen at the same time.

Although this seems a bit limiting, in some cases this solution is actually fine. When trying to get the number of packets that passed through one path, and the total amount of bytes, a very slight inconsistency between the two, or a value actually old by a few nanoseconds isn't much of a problem. In figure 4.19 this approach is referred as CounterMP. CounterMP is using the per_thread template to duplicate the two packets and bytes counter per thread, with care for alignment and false sharing as discussed in section 4.2.

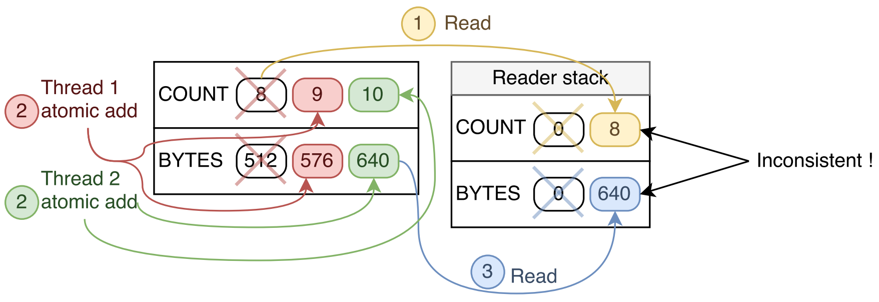


Figure 4.17: Update using atomic operations. Each individual counter is read atomically and is fresh at the time of reading, but both values cannot be read atomically and therefore may not be synchronized.

Word-consistent One other possibility is to use a variable that all threads will read and write using atomic operations. The atomic add operation guarantees that if multiple threads add some number to a variable at the same time, the value will be incremented with the sum of each thread's increment. To prevent the reader from reading some old data, the value is generally enclosed by memory barrier or declared as volatile. This protects against stale data, but the consistency is limited to the architecture word size. In the counter example shown in figure 4.17 the value of each individual packet and byte counters will be consistent and guaranteed to be fresh, but both are not synchronized. In figure 4.19 this is referred as CounterAtomic. The implementation simply uses 64 bits atomic operations to update both volatile 64bits packets and bytes variables. The atomic approach is therefore fine for variables that do not need to be synchronized. This can be used to count the number of packets dropped by a queue, to only count the number of packets an element handled, for reference counting, . . .

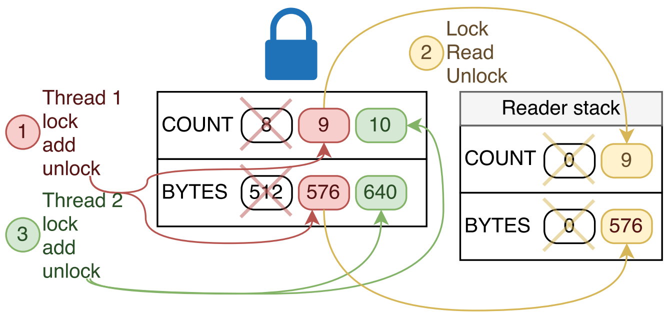


Figure 4.18: Protecting the mutable data using a lock. Before accessing the data, any reader or writer grabs the lock. This approaches provides full consistency of the data but does not allow concurrent writers.

Another solution is to protect the data-structure using a plain old spin-lock letting only one thread access the data at any time as shown in figure 4.17. In figure 4.19 this is referred as CounterLock.

Consistent A variant of the duplication solution is to duplicate the mutable data structure per-thread, along with a spinlock. The writers take their per-thread lock, modify the data and then release it. While the readers take the lock thread per thread read the value, and move on to the next thread. In figure 4.19 this approach is referred as CounterLockMP. This allows to ensure consistency when reading each individual per-thread bucket, but not to get a single consistent snapshot of the aggregate. To allow the per-thread approach for fresh and consistent data, one can follow the per-thread lock approach, and grab all the locks when a read must happen, before accessing individual buckets. In a write mostly context, the read method is not of interest, therefore we do not benchmark the fully constant approach in figure 4.19, but it will be referred later as CounterPLockMP.

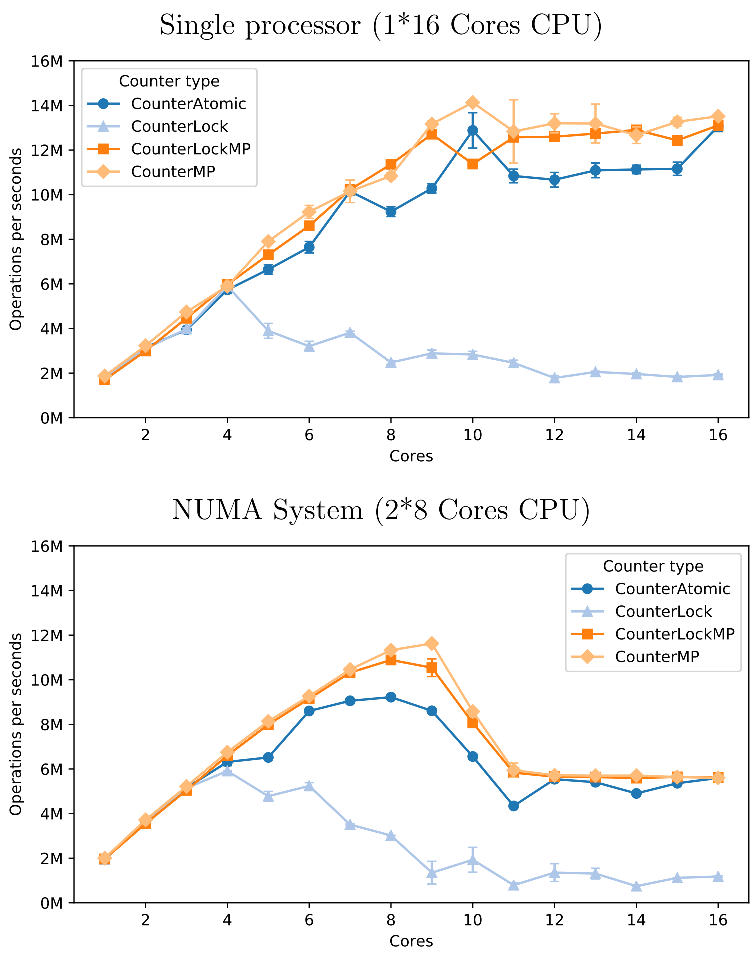


Figure 4.19: Performance of multiple data structures in write mostly situation. Counting the number of packets and total bytes passing by using various underlying data structures. Note that the *Counter* value is wrong, as it is not protected against multi-threading.

Evaluation Figure 4.19 compare all those approaches using a varying number of cores on a NUMA architecture and non-NUMA architecture. 64 bytes packets are generated in loop locally. They traverse the given counter implementation before being destroyed. The results are the average of 3 runs of 5 seconds each. The counter is only read at the end of the 5 seconds. The single processor system is a 16 cores Xeon E5-2682 v4 while the NUMA system has two 8 cores E5-2620 v4.

Without surprise the duplication approach performs better. However this works because in this situation it is not a problem if the count is not totally exact. Note that on the NUMA system, the memory allocation is not NUMA aware (that would need one more indirection level to split the structure per-NUMA node) and when using cores of both CPU the performance does not improve as much as when using more cores from the same CPU.

If the data structure must allow for an atomic “snapshot” of the per-variable state but not the whole structure, CounterAtomic offers more or less the same performances but with a lower memory footprint, and is sure to not read stale data. With this structure, the operator is sure to read the actual real number of packets or bytes but cannot ensure the correlation between both.

The per-thread lock approach does not impose any performance hit. As each threads has its own lock, there is actually no contention. One use case for fully atomic write mostly would be a rate element. To compute the rate, the element needs to remember the number of packets, the time it has seen the first one, and the time it has seen the last one. The rate computation needs the whole state to be consistent. If the number of packets is updated, then read by the aggregator but before the time of the last packet is updated the rate could be wrong.

The lock is the slowest solution as it never enables concurrent writers.

4.3.2 Read mostly

On the contrary, some elements *read mostly* mutable state. That is in fact the biggest use case when making the Click Modular Router thread-safe. When building a multi-threaded network function, there are in general a lot of configuration variables that are read by one or multiple threads for each packet but can be changed by any thread at any moment. Data caches are also examples of read-mostly structures, such as name servers cache or web servers serving static pages[78].

RCU An algorithm performing well under read mostly operation is the *read-copy-update (RCU)* data structure.

As shown in figure 4.20, RCU relies on a pointer mechanism. The only job a reader needs to do is to follow the pointer as shown in event 1 (with some limitations discussed later). When a writer wants to modify the data, it will copy the whole structure and update the copied version. The writer will then swap the pointer to the new copied data structure (event 2). Subsequent readers will therefore access the new version of the structure (event 3). At that time, two versions of the data exists as the first reader

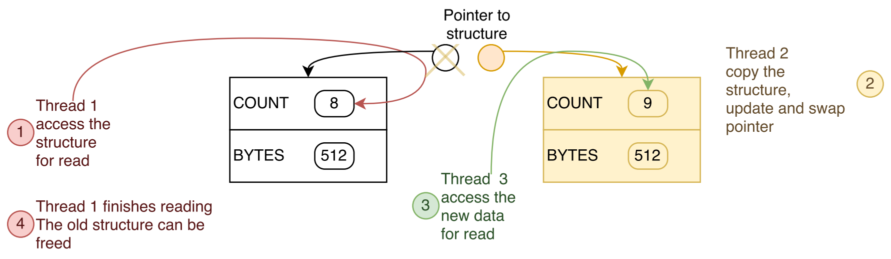

Pointer to structure
Thread 1 access the structure for read
1
COUNT
8
BYTES
512
Thread 2 copy the structure, update and swap pointer
2
COUNT
9
BYTES
512
Thread 3 access the new data for read
3
Thread 1 finishes reading The old structure can be freed
4

Figure 4.20: RCU order of events. A writer will copy the structure, update it and swap the RCU-protected pointer. Therefore multiple versions of the same data can live at the same time allowing the read operation to be very efficient.

is still accessing the previous version of the structure. When the first reader finally finishes reading, the old version can be destructed. The problem with RCU is to know when the last reader has finished reading and the old version of the structure can be reclaimed. While it would seem at first glance that simple reference counting can solve the problem, it is actually not a solution because a reader cannot grab the pointer and update the reference count atomically. A thread could reclaim the old structure after another reader has read the pointer but still has not updated the reference counter.

RCU is a structure used a lot in kernels to protect variables and linked list of mostly read data. But to solve the memory reclamation problem, the kernel RCU uses multiple facilities that are not directly available in userlevel such as disabling preemption or that would be too complex in userlevel such as scheduling itself through all CPUs. Therefore we will not review kernel-specific deferred destruction approaches.

As such, RCU is not a very known technique in the userlevel world. Although LibURCU[79], a userspace library that implements multiple RCU-like locking patterns similar to Linux ones enables to use RCU in userlevel. The faster of the multiple implementations they propose on the read side (that we seek to handle millions of packets/locks per second) is the *Quiescent State Based Reclamation(QSBR)*[80]. QSBR requires each thread to call periodically some function that declare a *quiescent state*. A quiescent state is a moment where a thread does not hold any reference to an RCU-protected structure. Moments between an update and the time where all threads passed through a quiescent state is called the grace period as shown in figure 4.21. When a thread swaps an RCU pointer, it knows the reference to the old data will be safe to delete when all threads have gone through a quiescent state, as afterwards they cannot have kept a reference to the old RCU data.

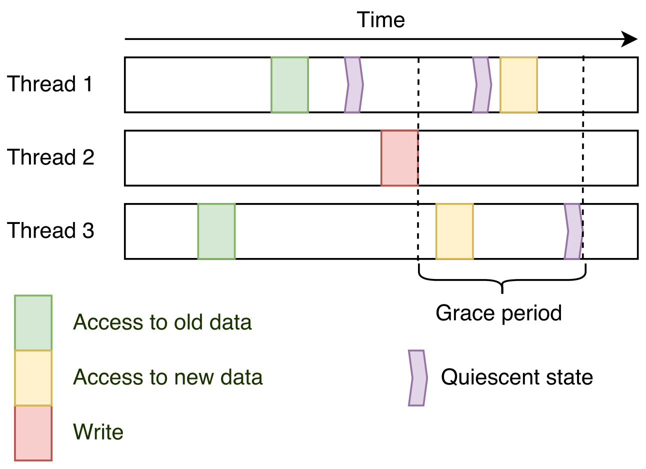


Figure 4.21: QSBR method to detect when an old version of an RCU-protected data is safe to delete. Each thread must pass through a quiescent state from time to time indicating it doesn't hold any RCU protected reference. When all threads have passed a quiescent state we know any prior reference is not held.

[81] reviews multiple userlevel RCU implementations, all relying on a global RCU system where moment between quiescent states may be long. In the context of a non-sleeping run-to-completion model it would introduce jitter when a thread is doing the garbage collection as the amount of work depends on a somehow unbounded usage, proportional to the number of elements used in the configuration and their usage of RCU.

Therefore we target a non-global (*i.e.* per structure or at most per-element) RCU that developers can use, allowing to bound the time a write will need to be in effect (*e.g.* for updating firewall rules), bound the time of writes synchronization (*e.g.* time for one write to be effective) and deferred destruction routine time.

We follow the idea of epoch-based reclamation[82], but differ in some points, particularly that our implementation is non-global for the reasons cited above. To avoid costly memory allocation and having to handle some kind of garbage collection we use a ring instead of allocated memory.

Writers increment an epoch number when finishing a write section. The only task of each reader when entering a read critical section is to copy that value to a per-thread

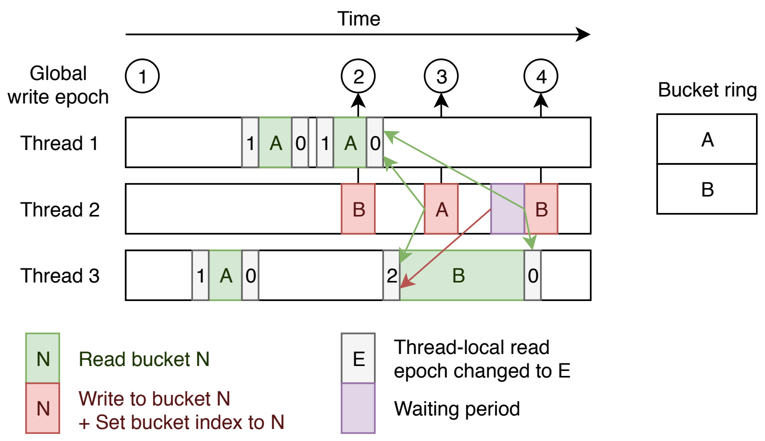


Figure 4.22: Our EBR, ring-based method to implement userlevel RCU

epoch number as shown in figure 4.22. The corresponding algorithm is algorithm 2. When the read critical section terminates, the read epoch is put back to 0. In practice an RCU<T> template is used to enclose the T structure with the write epoch variable, the per-thread read epoch vector, and create a ring of multiple T structures. The size of the ring is N, which is at least 2 to allow one writer to write in a bucket while readers access the other bucket. When a writer starts writing, the current ring bucket is copied to the next one, and the reference to that bucket returned to the caller. When the caller has finished writing, the epoch is incremented and the bucket index is updated as per algorithm 3. As writers advance the new bucket to use may wrap around the ring and try to overwrite a bucket currently accessed by readers. This will happen if multiple writes happen faster than a single read. The writer only needs to check that all read epoch are zero or above the write epoch minus N. If the epoch is 0, it means the thread is not currently reading any bucket. If the epoch is bigger than the write epoch minus N, it means that the current bucket of the given reader is accessing a bucket further than the bucket the writer is going to overwrite.

In the other cases (the epoch is less or equal to the write epoch minus N) the writer must wait. This situation is shown in figure 4.22 where the thread 2 executes two write operations during a single read of thread 3. Thread 3 cannot overwrite bucket B because it is still being read by the thread 3. It is detected because the thread 3 epoch is the write epoch minus N, which is $4 - 2 = 2$. The blocking time for a write is limited to the size of a read critical section around the very same structure. However the writer will only block when wrapping around the ring. At the price of a memory trade-off, the size of the ring can be increased up to a point where it is nearly impossible that multiple writers wrap around the ring faster than a single read critical section, meaning that the next bucket will always be overwritten without locking. In practice a value of 2 is usually enough because read critical sections are usually faster than write critical sections.

In most RCU implementations, a single writer is not directly enforced. Another locking facility such as a spinlock can be used to ensure that a writer does not ignore a previous write. Our version directly forces a single writer using a spinlock, as it is not naturally multiple-writer safe.

Algorithm 2 RCU read lock

```
function READ_BEGIN
    $readEpochs[thread] \leftarrow writeEpoch$
    $localCurrent \leftarrow currentIndex$
    $readBarrier()$
    return $\&ring[localCurrent]$
end function
function READ_END
    $readEpochs[thread] \leftarrow 0$
end function
```

Readers-writer locks RCU is not the only structure meant for read mostly. Multiple-readers, single-writer locks fall in the same category. They are special spinlocks that allow multiple readers at the same time, or one writer but not both. For comparison, two different implementations of Readers-Writer locks (RW) are also benchmarked. RW is the classical multiple-readers, single-writer lock based on atomic CAS instruction to allow either multiple readers or a single writer.

The PRW version is the Click original one (though we ported it to userlevel ourselves) which uses one spinlock per core, allowing to avoid the heavy CAS instruction and use a faster atomic swap instruction. But this makes the write much heavier as a writer must grab the locks of all cores as shown in figure 4.23.

Algorithm 3 RCU single writer lock

```
function WRITE_BEGIN
    $writeLock.acquire()$
    $nextCurrent \leftarrow (currentIndex + 1)\%N$
    $minEpoch \leftarrow writeEpoch - N$
    for $i$ in $n\_threads$ do
        while $readEpoch[i] <= minEpoch$ do
            relax cpu
        end while
    end for
    return $\&ring[localCurrent]$
end function
function WRITE_END
    $writebarrier()$
    $currentIndex \leftarrow nextCurrent$
    $writeBarrier()$
    $writeEpoch \leftarrow writeEpoch + 1$
    $writeLock.release()$
end function
```

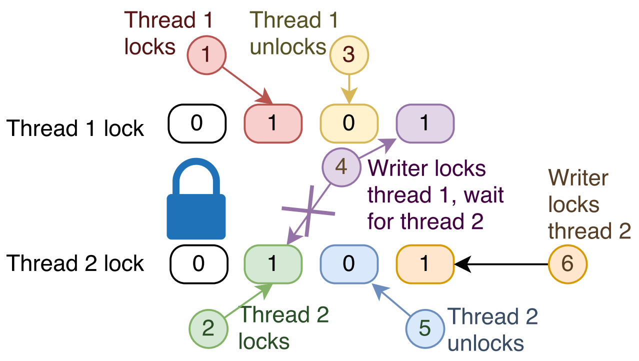
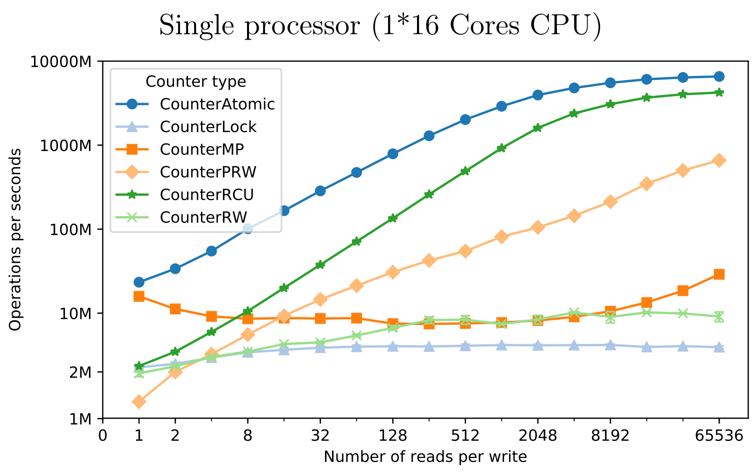


Figure 4.23: RW lock based on per-core spinlocks



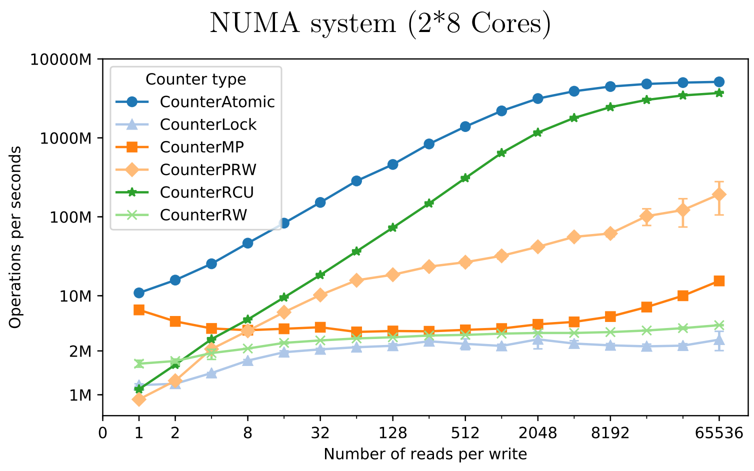


Figure 4.24: Number of reads (read 64 bits count and bytes count) or write (add packet count and number of bytes to the counter) per seconds with increasing read rate. Using a NUMA system (top) and a non-NUMA system (bottom) with 16 cores. Both axes use a log scale. 8 Cores version can be found in appendix B. Note that CounterAtomic and CounterMP do not allow consistent read of the structure.

Evaluation Figure 4.24 shows the same counter test case using 16 cores, but after each batch of packet counted (batches of 32 packets are generated), the values of the counters are read (the number of packets and the amount of bytes) at a varying proportion of reads per write, starting with 1 (alternate read/write) up to 65536 read operations in a loop after each write. This is not very realistic for a counter function, but does the job of benchmarking read mostly efficiency.

CounterAtomic and CounterMP do not allow consistent read of a structure and are somehow to put aside. If inconsistency can be allowed, then atomic read/write/add operations perform best as soon as reads are on a par with writes. The CounterMP approach which performed best in write mostly needs to aggregate all thread values for each read operation, seeing its performance de-gradate as the number of read per write increase.

If consistency is needed, the RCU data structure becomes the most performant when there are at least two reads per write on non-NUMA systems. On NUMA systems, when the number of cores is huge the RCU implementation relying on a per-thread epoch takes the inter-CPU hit, as it needs to read all per-thread epochs number and only performs better when there are 4 reads per writes when using 16 cores from two CPUs.

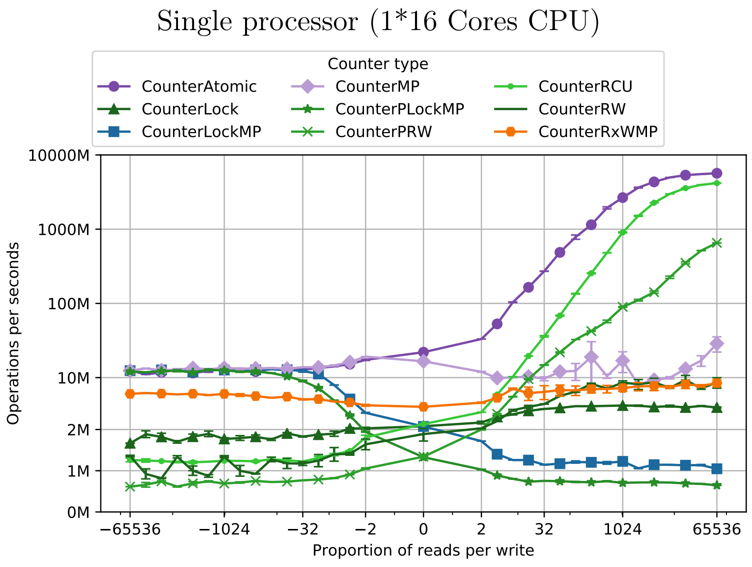
If the structure protected is huge, the RCU approach may begin to be slower as each write operation leads to a copy. The PRW lock approach is the better to use in that case, but still with reads around 1000 times slower than writes if they are a large majority (>=2048 reads per writes). The PRW lock performs much better than the one based on atomic CAS instruction (RW) as soon as the reads are prominent but performs awfully when there are less than 8 reads per write. This is expected as the writer needs to grab all per-thread read locks, repeating a costly atomic operation as many times as there are threads.

Appendix B also contains figures for varying number of read per write with 8 cores.

4.3.3 Update and degeneration

Figure 4.25 shows all data structures tested under a various proportions of reads and writes,1 from the write-mostly situation to the read-mostly situation. Ignore the CounterRxWMP for now. In overall, the CounterAtomic and CounterMP perform well, but they do not allow full consistency. One will quickly see that the central "update case" lacks an efficient data structure. Moreover, all data structure that performs well in read mostly perform badly in write mostly, and vice versa. Therefore we seek for a structure that would perform "correctly" in all cases, and performs well on average.

We combined the duplication approach with an exclusive multiple-readers multiple-writers lock (R xor W, or RxW). This class of locks allows either multiple readers or multiple writers but not a mix of the two. RxW is not a member of the class of classical Read-Write locks, that is multiple-readers single-writer locks because it allows concurrent writers. As the lock is used to protect a per-thread duplicated data structure, it is fine to allow multiple writers as they all access their own per-thread bucket. This



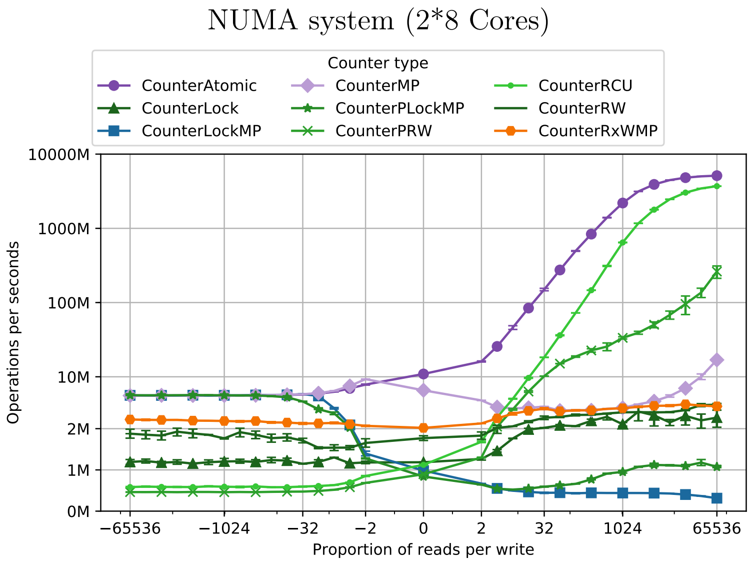


Figure 4.25: Performance of data structures of interest in terms of operations per seconds under a various amount of reads or writes. On the left, the graph shows the performance of the structures in a write-mostly situation. A value of -65536 means 65536 writes per read. This value decrease up to 0, meaning that read and writes are on a par. While on the opposite when the value reaches 65536, 65536 reads are executed per write. Using a NUMA system (top) and a non-NUMA system (bottom) with 16 cores. Both axes use a log scale. 8 Cores version can be found in Appendix B. Note that CounterAtomic and CounterMP do not allow consistent read of the structure, while CounterLockMP is not doing fresh reads.

ensures that when reading, none of the buckets are modified and therefore provide consistency and atomicity with the advantage of supporting multiple concurrent writers. The limitation of this data structure is that writers should not read other's thread values as other writers may be accessing them. They must release the write lock and grab a read one. The algorithm uses a *compare-and-swap (CAS)* instruction to change a unique integer value. The CAS is an atomic instruction that will compare a memory value with some expected value. If the value is the expected one, it is atomically swapped with another desired value. Therefore two CAS cannot interfere with each other and if done concurrently one of the two will fail. CAS is usually used when a value must be read from memory to a register, then modified in the register and finally updated in memory. The CAS will ensure that the memory value wasn't changed while the computation was done before the value is written back to memory. The lock is based on a volatile integer value. When positive, the integer is the number of readers accessing the data, and when negative it is the negative number of writers accessing the data. Readers are increasing the value if it is already positive or null using a CAS instruction to protect against concurrent modifications. Writers are decreasing the value if it is already negative or null, also using a CAS instruction. In other cases readers and writers are spin-looping. The CounterRxWMP line in figure 4.25 shows the performance of this approach. It keeps a nearly constant operation rate, while performing around 2 to 4 times better than the CounterLockMP (per-thread duplicated approach, with a lock also duplicated per-core).

Appendix B.0.1 presents a slightly modified version of the RxWMP that allows to "prefer" read or write. This version may be useful to ensure that either reads or writes will succeed as soon as possible, but does not improve the performance and, as such, is left in the appendix.

Table 4.1 shows a summary of the best basic structure to protect mutable data to use in each situation according to the level of consistency needed and the most prominent operation. The nuances highlighted in this section are to be kept in mind. Moreover, more complex structure for specific operations such as inserting an item in a linked list can perform faster. The micro-benchmark of this section is of course limited, and abusing the per_thread technique may hit shared layers of caches but we believe it is correct enough to give valuable insight on scaling data structures using non-sleeping facilities. The number of read per write should not be used as a direct approximation

	Read mostly	Update	Write mostly
Stale	Atomic	Atomic/MP	MP
Word-consistent	Atomic	Atomic/MP	Atomic
Consistent but stale	RCU	RxWMP	LockMP
Atomically consistent	RCU	RxWMP	PLockMP
Huge structure	PRW	RW	Lock

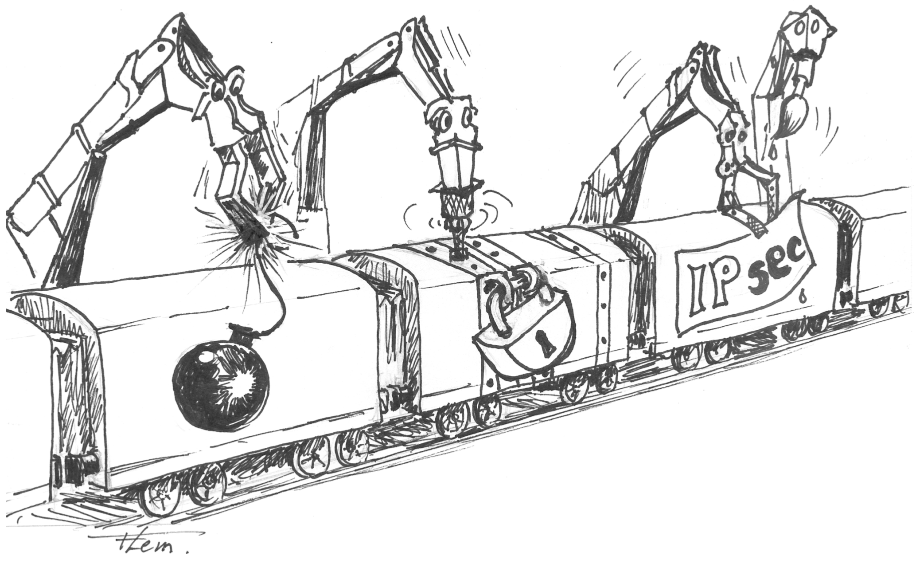
Table 4.1: Summary of the best structures to provide a certain amount of consistency (vertical axis) according to the most prominent operation (horizontal axis).

as in realistic cases all CPUs are not accessing the same structure at the same time, but more as a rough idea of when some "lock" mechanisms become faster than others.

Open Source Availability

All the datastructures presented in this chapter are available as C++ templates, to easily protect/duplicate a data structure per-thread. It is part of FastClick at [21].

Try it out ! ■



5

An NFV Dataplane

Chapters 3 and 4 focused on building a high performance, multithreaded platform, leading to FastClick, our improved version of the Click Modular Router. But FastClick is packet-based, and does not provide any efficient service chaining. *Network Function Virtualization (NFV)* applications and particularly middlebox ones need specific facilities such as the concept of flows and ways to temper them, *i.e.* they need to see a seamless stream of payload passing through the box.

In current designs, VNFs have difficulties to cope with the growing needs for more throughput because they are often independent systems working like complete black boxes, totally unable to cooperate between themselves.

Three typical implementations of a service chain are shown in figure 5.1. They all implement mostly the same logical chain, where the packets need to go through a firewall which blocks malicious traffic, an intrusion detection/prevention system (IDS/IPS), a vendor-specific application (*e.g.* proxy cache, content optimization, ad-removal or insertion, parental filtering, *etc.*), and finally go through services for the internal network such as a NAT or a load-balancer.

The first implementation is a standard Linux box implementing all functions using common software, Snort[83] for IDS, NetFilter/IPTables/NFT for the firewall, HAProxy[84] or NGINX[85] for load-balancing, ... This is the setup often found in small networks. It is **inexpensive** but also **slow** as we'll show in section 5.1.

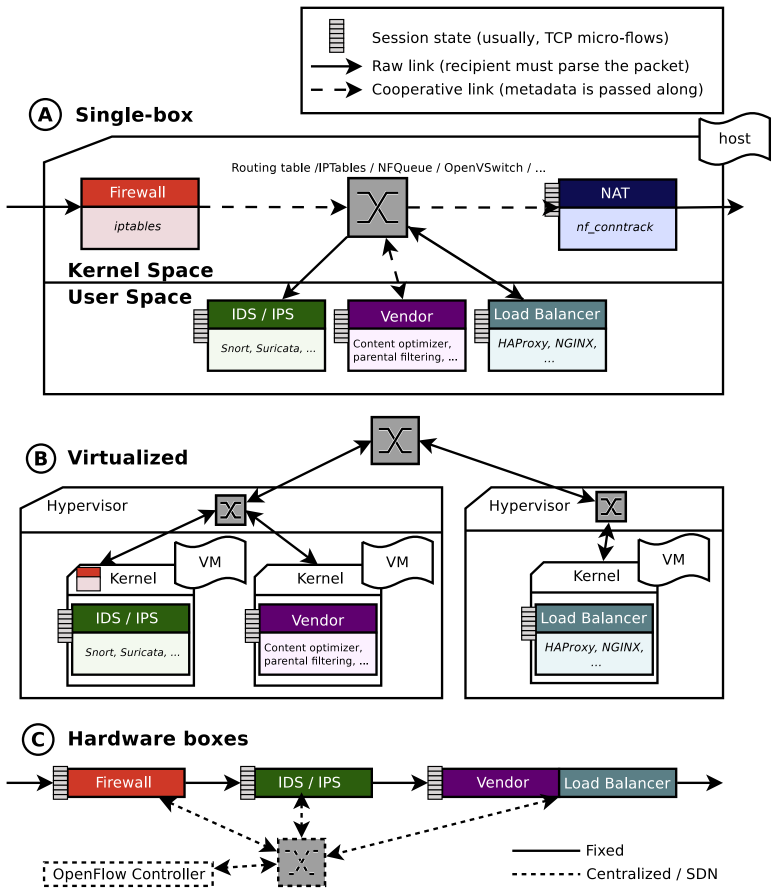


Figure 5.1: 3 different ways to build a middlebox service chain. On most links, there is no cooperation to avoid redundant operations.

The second one uses virtualisation and switches to dispatch packets between VMs containing mostly similar software. This setup is easier to **scale** as VMs can be replicated, potentially across multiple servers. The setup is also more **reliable** thanks to the virtualisation layer allowing to have fail-overs VMs on different servers that can be started when the original one crashes. But virtualization also introduces penalty in performances, although recent research papers try to reduce this hit[86, 87]. Making the network functions virtual also allows to **outsource** the network functionality to some datacenter instead of running a dedicated x86 infrastructure on campus.

The third chain, mostly seen on large campuses like ours uses different physical boxes to achieve the same results in hope of achieving **better latency and throughput**, but is also **expensive**.

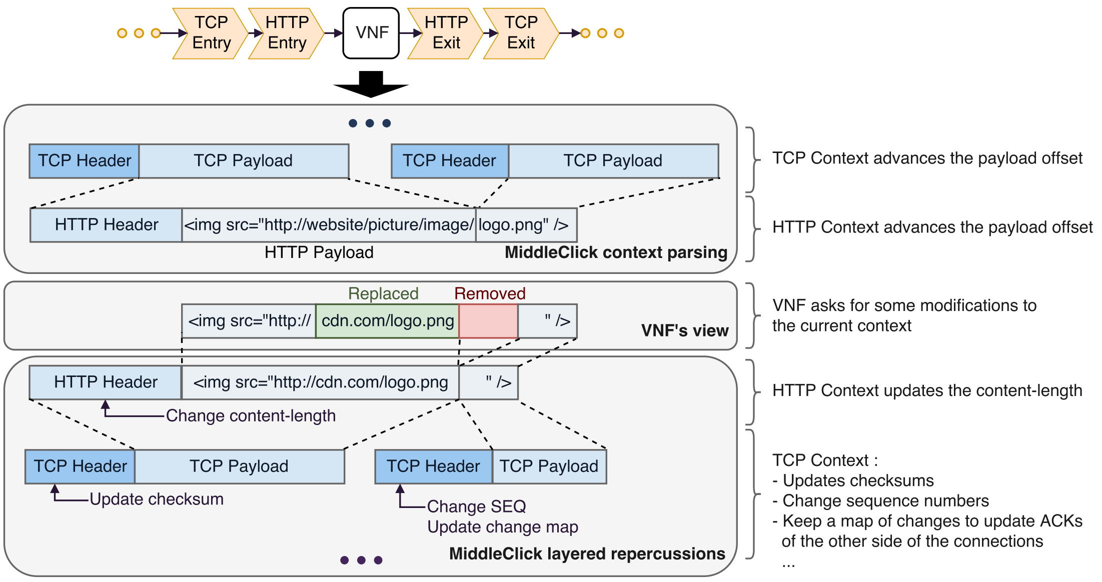

TCP Entry
HTTP Entry
VNF
HTTP Exit
TCP Exit
TCP Header
TCP Payload
HTTP Header
<img src="http://website/picture/image/logo.png" />
HTTP Payload
MiddleClick context parsing
TCP Context advances the payload offset
HTTP Context advances the payload offset
Replaced
Removed
<img src="http://
cdn.com/logo.png
" />
VNF's view
VNF asks for some modifications to the current context
<img src="http://cdn.com/logo.png
Change content-length
HTTP Context updates the content-length
Update checksum
Change SEQ
Update change map
MiddleClick layered repercussions
TCP Context :
- Updates checksums
- Change sequence numbers
- Keep a map of changes to update ACKs of the other side of the connections
...

Figure 5.2: MiddleClick flow abstraction system. A VNF can make requests to its current abstract context that takes care of the implications for the protocol it supports, and then pass the request to the lower context and so on. This allows to easily build support for tempering flows of new protocols on top of others.

Before network functions moved to software with the NFV trend, most middleboxes were implemented as hardware boxes. This **slow down innovation**, because upgrading the box to support new protocols means changing the box. Therefore the Internet is left with a lot of middleboxes that do not support newer protocols (*e.g.* IPv6) without owner willing to pay to replace them.

Of course the reality is not always like those exact setups, but they all share the same problems. Observations of usual middlebox service chains lead to the 3 following issues:

(a) A packet is partially or completely re-classified in each middlebox component, *i.e.* packet headers are inspected to classify the packet according to known values such as "destination port 80" to decide that a packet is HTTP. This is what we call the packet **traffic class** classification. Rules to classify packets according their traffic class are not limited to fields, they could be packets sharing some meta-characteristics such as a pair of ingress and egress routers.

(b) A dictionary data structure is present in all stateful middleboxes to assign a memory space for each group of packets belonging to the same **session**. The best known concept of session is the TCP 4-tuple, shared by all packets of the same TCP session, also sometimes referred to as TCP micro-flow. We reference the per-session unique space resulting from the mapping done in the dictionary data structure as the **scratchpad** of that session.

(c) The VNFs in the chain relies on slow OS capabilities such as a generic TCP stack that may be only partially needed and not designed for specific needs of middleboxes.

(d) Modification of a stream on-the-fly is usually implemented by terminating the connection using a server socket and re-opening a client connection towards the destination on behalf of the source, a very heavy process.

In this chapter we design and implement a prototype of a system in which the packets begin their journey through a unified flow manager responsible for the classification, which is then reused by all middleboxes. By enabling middlebox cooperation, they can receive packets with a given associated flow identifier instead of exchanging raw packets.

The flow manager also handles the sessions for each middlebox component, allowing to avoid multiple, often identical, hash tables along the way to find the session of each packet.

By unifying the traffic class classification and session mapping done in VNFs along a service chain, we ensure that each field of the packet is looked at only once for the same values, and the result of the classification and the session mapping are reused across all the following middleboxes.

The framework also provides a zero-copy **stream abstraction**, allowing to modify packets of a same session without the need for any knowledge of the underlying protocols. The abstraction enables building support for new protocols on top of others protocols easily. When an HTTP payload is modified, the content-length must be corrected. A layered approach allows to back-propagate the effect of stream modification across lower layers. Following this approach, we provide a TCP-in-the-middle stack which will modify, on-the-fly, sequence and acknowledgement numbers on both sides of the stream when the upper layer makes changes. This allows to modify the number of bytes in the stream without terminating the connection.

The system finally provides support for a mechanism to "wait for more data" when a middlebox needs to buffer packets, unable to make a decision while data is still missing. Our TCP-in-the-middle implementation supports pro-active ACKing to avoid stalling a flow while waiting for more data, and allows to handle a large amount of flows using a run to completion-or-buffer model which avoids costly context switches. While this model is asynchronous, it still offers the convenience of a blocking system and allows to handle very large numbers of concurrent sessions.

In our design, each flow element that needs more packets to process the session will buffer packets per-session in the per-session scratchpad and go back to the input loop until a packet of the same session arrives. The packet will be buffered if and only if a decision on the processing of the flow cannot be made right away, *e.g.* because an out-of-order packet is missing or we are in the middle of a potential matched pattern in an IPS, for instance.

The system provides services tailored to the service chain according to the minimal features needed for the VNFs composing the chain. The per-session state structure is minimized to fit all space needed for the network functions. The fit-for-all state structure avoids relying on dynamic memory allocation used by key-value stores like proposed by OpenBox[41] or a unique context pointer that must be shared by all applications such as proposed in mOS[46] that would need one more memory allocation per-NFs. Therefore our system combines the advantages of efficient but purely end-to-end and non-cooperative systems such as DPDK[31], Netmap[32], Arrakis[37], IX[59] or specific userlevel stacks [27, 45, 46, 55], and the contrasting approaches that build on reusing components such as CoMb[40], SNF[39] or OpenBox. Our design therefore combines efficient consolidation with tailored services. If no TCP reconstruction needs to happen for the VNFs along the service chain, the reconstruction does not happen, and if multiple VNF needs it, it is done only once. Section 5.3.1 studies deeper the state of the art and our specific contributions.

In section 5.2, we present our design to build an efficient NFV dataplane. Section 5.4 explains how we combine the VNFs among the service chain. From there stems a highly parallelisable and non-redundant stream architecture that can be used as a basis to support multiple protocols, as explained in section 5.5.

To emphasize that the prototype we built is only one possible implementation of the design, we defer all implementation related questions such as how a VNF can expose its classification and session specification up to section 5.6, where we explain how we built our prototype on top of FastClick, that we called MiddleClick.

Finally, we evaluate the performance of the prototype in section 5.7 that results in extremely fast middlebox service chains that achieve, to the best of our knowledge and the largest review of the state of the art we could achieve, unprecedented speed for pure single-box software implementation and possibly even better results when cooperating with heterogeneous hardware and multiple boxes, which is the subject of chapter 6.

This chapter in a nutshell

▶ **Context of this chapter**

- While basic high-speed platforms are mature, NFV platforms are still problematic, lacking efficient session management able to handle large amounts of flows
- Along a service chain, packets are re-classified multiple times, sessions are rebuilt, and most VNFs rely on slow OS-provided mechanism
- Current solutions tend to fall under two categories: decompose middleboxes to re-use and/or consolidate components, or push further the end-to-end by bypassing the OS completely or implementing a full TCP-stack which, at the opposite prevents any consolidation

▶ **Highlight of our main contributions in this chapter**

We build MiddleClick, a new NFV platform on top of FastClick.

- MiddleClick combines the *static* classification done inside each VNF to allow for an offloadable and ahead-of-time minimized classification step

- Each VNF declares the session it wants to see associated with each packet and the per-session space it needs (*e.g.* a counter per IP pairs and TCP sessions). This allows for a generic and very efficient per-flow metadata, with a flexible definition of flow.

- MiddleClick combines the static classification with the dynamic sessions, dropping impossible paths and ensuring that session fields are not looked at if the path to the session actually fixes some values. *E.g.* an HTTP VNF may actually only need a 2 tuple as the IP destination and port may be fixed by the path to reach that VNF. Contrary to previous work, this would automatically be found as the fact of a low-level factorization and not user using specific pipelines of fixed protocols such as TCP.

- MiddleClick uses sessions to build multiple, generic *contexts* such as IP or TCP context. The contexts are layered, allowing to quickly use protocol on top of others, passing requests such as adding bytes from layer to layer. This allows very quick implementation of new protocols, with flexible underlying flow definition. *E.g.* The HTTP layer will change the content length, pass the request to the TCP layer that will change SEQs and ACKs for the packet and subsequent packets, ...We provide an efficient TCP stack for in-the-middle modification of flows.

- Based on the session and the context, MiddleClick provides a flow abstraction allowing to work on a stream of bytes, stopping current work when the VNF needs more payload without context switching. VNFs are still allowed to go down and check on packets if they asked for a stream of bytes, as all IDS do.

- MiddleClick allows to only instantiate the service VNF needs for each service chain. The model allows for a full TCP stack support that reuses tailored components according to the needs of each VNF and bring up only the minimal functionalities to serve the service chain, and not more.

- MiddleClick is a real, publicly available implementation. There are few NFV platform supporting flow tempering entirely available where the actual low-level problems such as how to classify flexible sessions for each VNF, provide a per-flow metadata map that can handle millions of flows is discussed and actually implemented. A lot of current works have blurred lines between what is actually done and what is conceptually allowed by their proposed design.

■

Contribution notice

Most of this chapter has been presented as an EuroSys'18 poster, and then as an invited paper presented to HPSR'18[88]. It has been done in collaboration with Cyril Soldani, Romain Gaillard and Laurent Mathy. Romain has specifically worked on section 5.5.5 as part of his Master Thesis[89], the TCP part on top of the flow abstraction.

One man cannot solve all the world's problems ■

5.1 Motivation experiment

While some middleboxes like port-based firewalls are typically packet-oriented, more and more middleboxes need to understand flows instead of packets. Examples of such middleboxes are deep packet inspectors (DPI) or IDSes, where attacks could span across multiple packets and one often needs to account for some relations between packets, or accelerators like proxy caches.

Some load balancing software like HAProxy[84] or NGINX[85] (when used as such) use the OS TCP stack, terminating connection and creating a new one even when it is not strictly necessary. However, [32, 45, 54] show that performance and scalability of the operating-system sockets are very limited. In short, most OS kernels such as Linux are great systems for *generic* network functions, but the interrupt system and the very big data structures such as the `sk_buffs` are not lightweight enough. As such, some software like SNORT[83] or Suricata[90] use the RAW mode for sockets, or specific I/O frameworks to receive raw packets, and do the classification of the packets into multiple flows and sessions themselves. However, this prevents efficient chaining of middleboxes as the operating system or the I/O framework gives raw packets to the first middlebox and only understands that it receives back some raw packets from it. A second pipelined middlebox will have to re-do all that classification again.

As a motivation experiment, we ran multiple service chains using Linux kernel facilities to run a simple L3 Router, a NAT and an IPS with Snort[83]. We profiled the CPU during the test using the Linux kernel perf tool[91]. We built a mapping of every function seen for more than 0.1% of the total CPU time share to 10 classes such as filtering, routing or flow management as shown in figure 5.3. The profiling is not exact and is limited in multiple ways. Profiling is known to miss some hit points, moreover

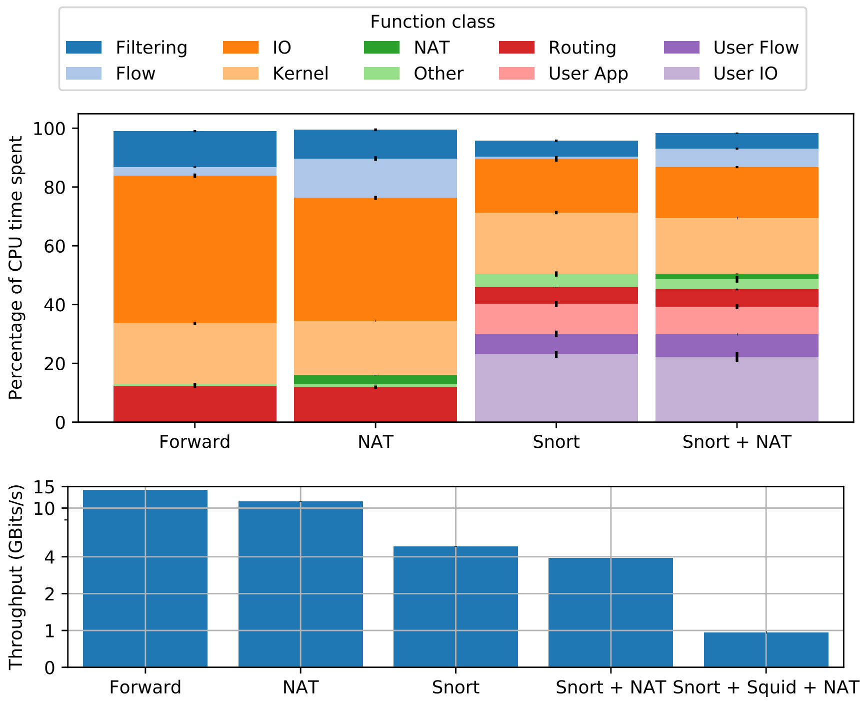


Figure 5.3: Profiling of multiple service chains running on a unique CPU core acting on 8K HTTP requests and responses. CPU time spent is aggregated in classes with a manual mapping from functions to classes.

the mapping is not exact as the assignment of functions to classes is somehow subjective. But it will give an insight into how much performance could be leveraged from better chaining and re-using previous classification. In our experiment, Snort does not actually execute any rule but still launches the pattern matching algorithm and basic hard-wired checks on the packets. More information on the testbed can be found in section 5.7.

The first forwarding service chain (only a two entry routing table) spends around 50% of the time in the kernel I/O path and 10% of the time in routing (respectively the *IO* and *Routing* classes). The kernel path still involves some filtering hooks even if unused, explaining around 10% of the time spent in *Filtering* functions and the relatively high overhead of the *Routing* class even with nearly empty iptables rules and routing tables. The *Kernel* functions class corresponds to kernel facilities that cannot be tied to a specific function such as memory allocation or spinlocks, but are somehow proportional to the usage of kernel facilities. In the case of the first forwarding service chain, we manually searched the functions call graph to find out that most of the Kernel class is tied to IO.

The second service chain adds a NAT after the forwarding. The connection tracking (the *Flow* class) takes around 20% of the CPU time (in Linux, that is the nf_conntrack facilities) and a few percents for the packet rewriting itself (*NAT* class).

When adding Snort to the service chain, the CPU completely re-do a flow classification (*User Flow* class) that takes more or less the same time as the connection tracking done inside the Kernel, as used by the NAT. In the end, both do more or less the same work. Moreover, a lot of time is spent in *User IO*, that is the service chaining itself, passing the packets from the routing to Snort, and then back to the NAT. The *User IO* class also comprises parsing done inside Snort, that uses a RAW socket as explained above. A fair part of the work done in the *User IO* class was in fact already done in the *IO* class but is re-done again in Snort. Re-doing the same parsing could be avoided by using a unified classification system that allows remembering the classification of the packets when piping applications together. The *User IO* class could also be reduced using a shared memory system between the multiple tenants in the service chain, or run multiple functionalities inside the same application. One of the two flow parsings could be avoided while the time spent in filtering and routing could be reduced using a system optimized based on the service chain, tailored in function of the needs of the NFVs components. We will develop this idea further in section 5.2. In the end, the useful time spent for the full service chain is limited to one of the two flow classifications (*Flow* class), the Snort matching algorithm (*User App* class) and the NAT rewriting (*NAT* class), that is around 20% of the total CPU time spent, without accounting for packet I/O. As shown in chapter 2, the IO and Kernel class could be reduced by an order of magnitude by using a faster raw I/O framework such as DPDK[31].

Figure 5.3 also shows the throughput when adding the Squid[92] proxy cache to the full chain, but it is not profiled as the simple mapping from functions to classes would require too much work. Squid brings-in a third flow classification as it uses Kernel provided TCP stream sockets, bringing up a new in-kernel classification. And it actually does a fourth one in userlevel to map each HTTP session to a control block internally. Manual profiling of Squid also showed that it spends a lot of time in copying the stream content and in memory allocations, bringing up the case for zero-copy stream modification (without losing a socket "stream" abstraction).

5.2 Architecture for an efficient NFV platform

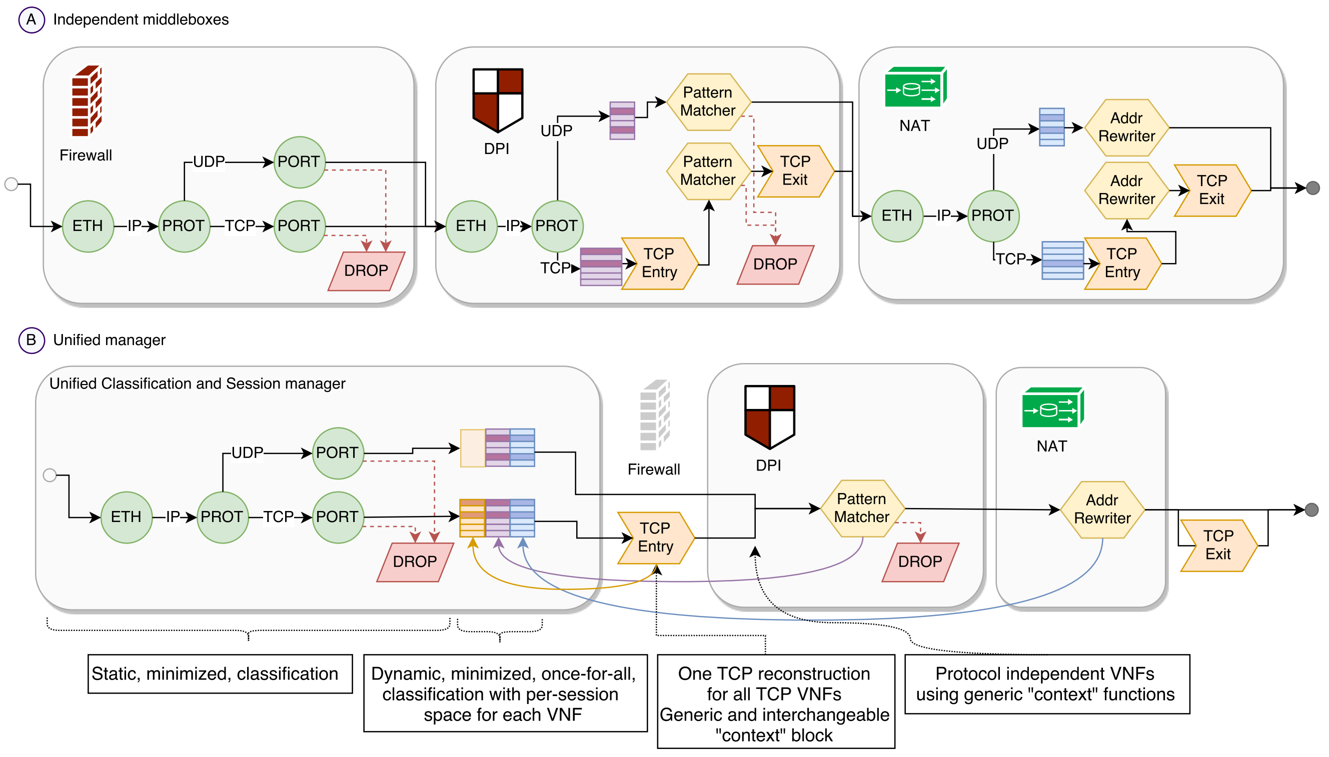

A Independent middleboxes
Firewall
ETH
IP
PROT
UDP
PORT
TCP
PORT
DROP
DPI
ETH
IP
PROT
UDP
TCP
Pattern Matcher
Pattern Matcher
TCP Exit
TCP Entry
DROP
NAT
ETH
IP
PROT
UDP
TCP
Addr Rewriter
Addr Rewriter
TCP Exit
TCP Entry
B Unified manager
Unified Classification and Session manager
ETH
IP
PROT
UDP
PORT
TCP
PORT
DROP
Firewall
TCP Entry
DPI
Pattern Matcher
DROP
NAT
Addr Rewriter
TCP Exit
Static, minimized, classification
Dynamic, minimized, once-for-all, classification with per-session space for each VNF
One TCP reconstruction for all TCP VNFs Generic and interchangeable "context" block
Protocol independent VNFs using generic "context" functions

Figure 5.4: Overall schematic of the architecture we propose as opposed to independent middleboxes

Any middlebox can be viewed as a set of simple components such as NAT handlers, pattern matchers, routing, ARP handlers, *etc.* Together, they are grouped to form a middlebox. *E.g.*, the Snort IDS uses a layered decoder approach to classify the packets into flows, a preprocessor to reconcile sessions of related packets, and apply the correct pattern matcher components accordingly.

Each component declares the kinds of packets it wants (*e.g.* HTTP packets, all TCP packets, ...), and if needed the session definition they want to see (group packets by IP pair, by the TCP 4 tuples, . . .), and in this case the size of the per-session scratchpad they need. Exposing classification and session definition along with a few protocol-specific needs does not impose any constraints on the developer, on the contrary it allows to remove any classification and session mapping from the application. The information provided by each component of the service chain is used to derive a unique classification table that will avoid further classification on the packet header. The classification runs before all components, as shown in figure 5.4 - B. The classification can therefore potentially be offloaded to some specific hardware or use classification functionalities of the NIC. The classification also finds a *Flow Control Block (FCB)* for each session at the same time. The FCB contains a session scratch pad big enough for each component. The scratch pad size is derived from the size that each component declares it needs, allowing to use a very efficient pool-based allocation and associate static offsets for each component that map into the scratchpad space of the FCB.

Instead of letting each VNF handle flow reconstruction and more generically protocol-dependent bookkeeping, we propose a context-based approach. In our design, packets flow through the VNFs as batches of packets of the same session. Special components change the context of the current session. *E.g.* from IP context to TCP context, handling protocol specifics at the same time. Each VNF can interrogate the current context of the batch to know the state of the flow, what is the offset to the payload in the packet, remove or add bytes without terminating the connection, buffer the content and ask for more data, and other queries that allow protocol-independent actions that will be further discussed in section 5.5.1. The context is not left between middleboxes avoiding for example multiple TCP reconstructions. We propose multiple abstractions on top of the context system. The simplest one gives the developer the batch of packets from the same session and a pointer to the session scratchpad space in the FCB that is unique for the current session. The most advanced one abstracts an event-driven handling of the flow, giving a developer an iterator to seamlessly iterate over the payload of the packets, relative to the current context (*e.g.* iterate over TCP payloads if in TCP context, or only the HTTP data without the header if in HTTP context).

Sessions can only be shared by components inside the same *memory space*, as sharing the state of the session across memory boundaries would need some kind of protocol which would probably defeat the purpose of factorizing the handling of sessions for another protocol. However, we will see in section 6.1 that we can still use session information from a previous memory space to accelerate the session handling in the next memory space, even between two separate computers.

5.2.1 Execution model

In the *run-to-completion* model, multiple parallel CPU cores process packets until they reach the output NIC. The packets never switch core and achieve a better data locality, at the expense of more instruction cache misses. Then the CPU core goes back to the input NIC to classify the next available packets in the NIC queue and repeats the cycle. Section 4.1 already reviewed the concept and pitfalls of the model.

We introduce an extended version of the run-to-completion model that we call the **run-to-completion-or-buffer** execution model. Each middlebox component can decide if it needs to wait for more data (*i.e.* packets) before proceeding, and save its state and the list of buffered packets in the per-session scratchpad. Therefore, the system runs to completion, or buffers the packet on a per-session basis but only if need be.

The run-to-completion-or-buffer model avoids the use of blocking threads as with standard sockets. In a usual socket implementation, a call to `read()` for packet data may block until some packets are available. In practice, high-speed NFV applications do not rely on blocking `read()`, and rather put the packets to buffer into a flow table. In our design, when there is no more data, the scratchpad can be used to remember the current state of the component, without using any other data structure to remember current active sessions. When a new packet of the same session arrives or a timeout occurs, the state can be recovered from the scratchpad directly.

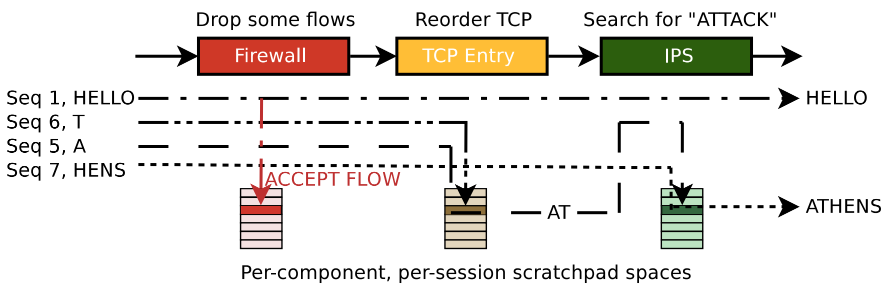


Figure 5.5: Scenario using the run-to-completion-or-buffer model

Figure 5.5 shows a scenario involving the run-to-completion-or-buffer model. It runs a service chain comprised of a firewall, a TCP reorderer and an IPS searching for the "ATTACK" word in flows. A first packet containing "HELLO" is sent. The firewall establishes, through classification that the packet is from an allowed flow and remembers the decision in its session scratchpad space. As that space is part of the unified flow table, it doesn't present much cost to remember the decision per-flow instead of traversing the classification rules for each packets. As a flow table will be needed for the truly stateful functions anyway, the firewall can ask the manager for a little more space in the flow table to remember its decision. The firewall then lets the packet go through, in this case running-to-completion. A second but out-of-order packet containing “T” arrives and is kept in the TCP reorderer session scratchpad, in this case running-to-buffer. When the missing packet with “A” arrives, the TCP reorder lets the two packets containing “AT” go through, now in order. While standard IPS would let the packets go through as “AT” does not fully match “ATTACK”, we propose a “True IPS” mode. The “True IPS” sees that it may be the beginning of the “ATTACK” word it is searching for, and keeps the packets in the session scratchpad until the end of the word arrives. In the example, when the last packet "HENS" arrives, the IPS lets the 3 packets go through as “ATHENS” is finally not matching “ATTACK”.

Note that stalling TCP packets have multiple implications that will be discussed further in section 5.5.5. This service is especially interesting for filtering content using machine learning or score-based anti-spam. The confidence in the fact that the content is not appropriate and should be blocked may arrive after the word has gone through. Traditional IPS such as Snort or Suricata (when used in IPS or “inline” mode) use a window over the stream of bytes from the same session. They wait for a certain amount of bytes and launch the inspection over the chunk when filled, and then start a new window. Therefore if an attack is done across two windows, it will not be detected. The newer Snort 3 allows to use protocol aware flushing and randomize the window size to cut the flow at more appropriate points or at least in an unpredictable way, but still suffers from the problem and allows for potential eviction of the IPS. Moreover pre-ACK window-based systems introduce jitter as they buffer multiple packets and flush them in one go. Our system will stall packets only if they may be part of a potentially problematic flow and will only force to flush them after some configurable amount of memory is reached, but keeping the state of the DFA in the scratchpad, preventing eviction.

Still, very specific use cases which need to maximize instruction cache hit can benefit from a pipeline approach as opposed to a run-to-completion approach as discussed in section 4.1. In this case, a link to the current flow and session must be passed with the packet in the software queues used to exchange packets between cores, with all the special care needed for objects shared by multiple threads.

5.3 State of the art

5.3.1 I/O Frameworks and virtualization

Many previous works have tried to tackle the problem of performance in VNFs, from different perspectives. One could argue that efficient service chaining is the problem of the Operating System, but generic OSes have proven to be far too slow for raw I/O [54] and middlebox implementations[46], a fact verified in sections 6.1 and 5.7 as well as the motivation experiment above. Previous work such as IX[59], or Arrakis[37] modified or re-designed operating systems to improve performance and isolation between middlebox components. Mirage[93] , NetVM[74] and ClickOS[94] try to make the components themselves faster using fast-deployable and efficient unikernels or light VMs but all three lack support for cooperation between components inside the OS and cooperation between multiple instances of the dataplane. NetBricks[42] reduces the set of core modules, allowing to ensure isolation at run-time using Rust that allows safe programming instead of relying on virtualization techniques. Our work is somehow orthogonal to those, NetBricks(Compiler-based isolation), ClickOS (Xen-based), NetVM (KVM), OpenNetVM[95] (container-based variant of NetVM) could be used under our framework to provide isolation between multiple set of VNFs. In other terms they provide an alternative to pure bare-metal FastClick as presented in section 3.2, more or less taking advantage of techniques up to section 3.2.5 such as zero-copy or multi-queuing. Section 5.4.3 discusses how to ensure memory isolation when the flow manager runs in a hypervisor or inside the operating system kernel, two situations that need to pass the flow information across isolated memory spaces. Chapter 6 reviews how to keep classification information across different boxes, or isolated environment that would be introduced by the isolation provided by those frameworks and their virtualization techniques.

5.3.2 Userlevel TCP stacks

CliMB[27] introduces a full event-driven TCP-stack inside the Click Modular Router[11] which has proven to be a good platform for middlebox and NFV implementations and helps to bridge the gaps, along with other user-space stacks[34, 45, 55], towards a full userlevel service chain completely by-passing the kernel. Yet, they are not in the scope of cooperation between instances of those stacks. They all receive and send raw packets, that would need full re-classification and protocol specific management for each VNF along the chain.

5.3.3 NFV Dataplanes

Flurries[96] use short-lived VNFs on top of OpenNetVM. Their system proposes the use of one (light) VM per flow with quick state cleaning of pre-allocated docker VMs. This allows a per-flow isolation but the performance drops quickly as the number of flow increases.

xOMB[38] provides better programmability by decoupling middlebox functionalities, and allowing to build simple pipeline of middleboxes functions. CoMb[40] explores consolidation of middleboxes for better resource management and reduces the need for over-provisioning. Like in our design, CoMb supports memory sharing with older applications that cannot be modified to take advantage of the facilities provided. But both those piece of work lack consolidation inside the low-level components, *e.g.* passing flows between applications and providing support to build functions on top of a flow abstraction that can be unified, likely leading to a limited throughput or a higher latency when the service chain is long. xOMB relies on heavy message buffers passed between components and their result is hardly achieving high-speed. For a function similar to the one evaluated in 5.7.3 their performance is more than a hundred times slower than our system. This difference of performance cannot only be the fact of their older testbed. Moreover xOMB does not provide a way to decompose and recompose multiple VNFs to consolidate the components of multiple middleboxes together and it is likely their performances will drop even further with a higher number of VNFs. As xOMB is not publicly available, it is not possible to dig into more up-to-date performance further.

NFP[97] automatically builds a parallel graph according to the order and dependency between VNFs and takes care of efficiently copying packets and merging them back. It unifies the static service chain classification, but not the intra-VNF classification, nor the dynamic flow tables and flow abstraction. While we assume a run-to-completion model in this work, their pipelining technique could be used in conjunction to our proposal.

5.3.4 Controller-based approach

SIMPLE[98] uses SDN to steer the traffic and simplify middlebox service chain definition. OpenMB[99] controls the service chain, and handle migration by allowing middleboxes to expose their state using an API. OpenBox[41] consolidates low-level functions across the service chain, using a controller to manage Click-based low-level components (though this is only a proof-of-concept and they allow for other implementations including hardware ones), extracting packet header classification to a unified parser but no stream abstraction or re-use of session parsing, *e.g.* sharing 5-tuple sessions which would be re-classified in each component. The OpenBox protocol already defines a per-session key/value store but it seems to be only conceptual at this stage, therefore many questions are not addressed such as how to choose the right data structure to allocate millions of per-session, per-VNF metadata spaces each second. Neither how do they address the recycling of that structure or how to handle possibly multiple levels of sessions (per destination, per ip pair, per TCP session, ...). We address those issues in this work. Those 3 solutions do not go as deep as providing highly parallelisable flow-aware processing. They do not provide an abstraction to implement middlebox functions that act *in-the-middle* to temper streams of protocols such as TCP, or tackle the issue of cooperation between multiple VNFs.

While our implementation can be used to manage and accelerate other applications like Snort using DPDK Rings or DPDK *Kernel Native Interface (KNI)*[1], our solution lacks an intelligent controller like in OpenBox[41], OpenMB[99] or xOMB[38]. A controller would allow to automatically combine the flow classifier tables and pass information between different boxes using tagging as proposed in chapter 6. However, we believe integration of our design into any existing controller would not be complicated.

[1]DPDK KNI allows to create a virtual interface that appears as a normal NIC to the Linux Kernel stack. But it is actually tied to a ring of DPDK buffers. It is a fast was to exchange packets between a DPDK application and the Kernel network stack

There are not many modifications to the core of Click, and many previous piece of work such as OpenBox already use Click as their data plane.

5.3.5 Graph consolidation

OpenBox, SNF[39] and CoMb do optimize the graph to reuse some basic components. CoMb decouples functions into common parts to allow to re-use some bricks. OpenBox goes a step further by differentiating kinds of bricks and re-order them - *when allowed* - to allow further merging of some bricks. SNF associates functions to more basic read and write blocks, allowing to synthesise the service chain graph even further. In our design, functions re-ordering is not automatic but classification is unified and back-propagated so that even classification steps to be done after some rewriting elements can be optimized to suppress unreachable paths or speed up classification of values that will be known in some part of the path.

5.3.6 Flow tempering

NetBricks[42] and mOS[46] both implement a TCP stack with some similar abstractions, but do not provide any factorization and acceleration of the full service chain. Their flow abstraction is limited to a less flexible window system and does not provide a generic non-TCP stream abstraction nor the session scratchpad facility, likely losing a lot of performances when the box actually runs many different VNFs. mOS offers a nice, high-performance event-based TCP stack. Most of the events they offer can be abstracted by a layer on top of a MiddleClick context element in a few lines as discussed in section 5.6.5, with the advantage that it can work for any protocol.More importantly, none of them proposes a way to modify flows (*i.e.* more than simple rewriting) without fully terminating the connection and therefore bringing up a lot of book-keeping. The system we propose lets endpoints handle the TCP semantics, making it future proof whereas most of the state of the art implementing TCP stacks only support part of the TCP protocol, strip options, or represent so much maintenance work that they are already abandoned. TCP Splice[100] avoids full termination by mapping sequence and acknowledgement numbers with a constant offset, but do not allow modifications. Pattent [101] describes a similar on-the-fly TCP modification, but leaves out certain details and do not take the same approach for retransmissions. They do not focus on service chaining, do not use shared flow tables, or do not describe classifications problems as we do in this chapter. Moreover our proposal is protocol agnostic, we allow a context-based system that allows to support *e.g.* HTTP modifications on top of TCP modifications.

The novelty of our proposal also resides in the fact that it is able to automatically collect information from the middlebox components and provides just enough, tailored services. MiddleClick computes a minimal state, called the *Flow Control Block (FCB)*, with offsets that are placed in predictable locations for fast lookups across VNFs. Only the needed protocol layers are invoked according to the context of each component, and, when possible, the layer is not left across middleboxes, *i.e.* the stream is kept and checksums, protocols specifics such as ACKs numbers are only recomputed once for all middleboxes. This combines the speed gains from reusing components like CoMb proposes, with the speed given by userlevel stacks like mTCP or CliMB or even more end-to-end systems like mOS, NetBricks, IX or Arrakis, allowing to even optimize the stack further by deactivating unused features, *e.g.* read-only TCP reconstruction for IDS services is more lightweight than modifiable TCP flows, which is in turn more lightweight than resizeable TCP flows that implies a lot of tracking facilities to correct SEQ and corresponding ACK numbers, which is still lighter than a full, connection-terminating TCP stack.
Compared with the state of the art, we also propose here a much more low-level drive into building a fast NFV dataplane agent that supports efficient service chaining.

5.4 Combining middleboxes

In the architecture we designed, each independent component defines the kind of flows it wants to receive. When middleboxes are chained, our system can take advantage of what each component needs to re-classify more lightly between each of the components and pipe them together more efficiently.

Note that flows are not limited to packet fields. *E.g.*, for an ISP a flow could be defined as packets from one ingress to a specified egress, the combination of the two determining the processing to apply. To avoid confusion, we'll refer to the flows in the sense of kind of data, defining the path packets from the same flow must follow as the traffic class. While we'll reference as packets from the same session the packets from

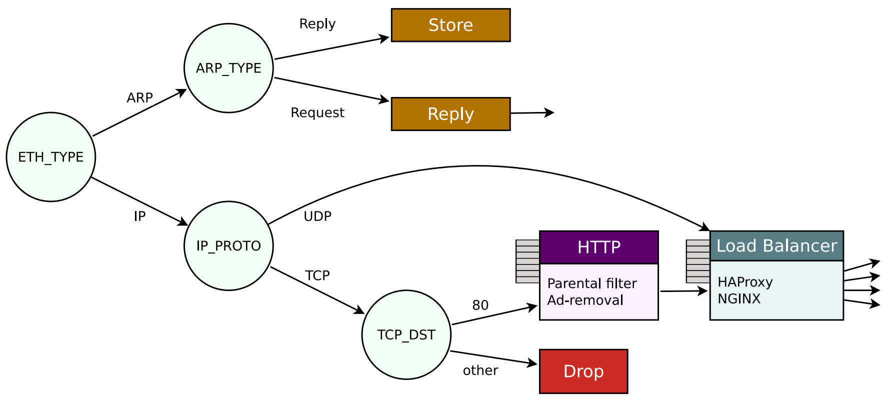


Figure 5.6: A small system handling ARP packets and applying some processing on HTTP traffic, dropping other TCP streams. It then load-balances UDP and HTTP traffic to some servers.

the same traffic class that share some properties, *e.g.* packets with the same 5-tuples that form a TCP micro-flow.

Consider figure 5.6 as a simplified example. This system handles ARP requests and replies in a packet-based fashion. It runs a per-session load-balancer for UDP and TCP traffic, but before that, passes HTTP (TCP packets with port destination 80) traffic through some HTTP filter (*e.g.* a parental filter or ad-remover). Other TCP traffic is dropped.

Each component must declare a list of packet types (*i.e.* a list of traffic classes) they want to receive.

The ARP subsystem wants to receive the traffic class of “ARP requests” and “ARP replies”, and needs no session management. The HTTP filter needs TCP packets directed to port 80. The load balancer receives any TCP or UDP traffic (though, given the wiring it will actually only receives some HTTP TCP traffic).

In figure 5.6, each circle is one step of the classification which would usually be handled by reading the corresponding packet fields to decide the next step when the packet reaches that point. The first level has 2 outputs, ARP or IP packets that could be identified as 1 and 2. Following this idea, reaching a leaf of the classification path can be considered as following a list of *next hop* numbers. Table 5.1 shows the resulting flow table (consider PFP entries in the table as wildcards for now).

To each rule will correspond a *Flow Control Block* (FCB), initially one per traffic class. Instead of classifying packets at all steps of the processing chain as with the

Rules							Flow Control Block	
Ethertype	ARP Type	Proto	Dport	Sport	Dst	Src	Next hops	Session space
ARP	Request	*	*	*	*	*	1, 2	-
ARP	Reply	*	*	*	*	*	1, 1	-
IP	*	TCP	80	PFP	PFP	PFP	2, 2, 1	TCPSession, int
IP	*	UDP	PFP	PFP	PFP	PFP	2, 1	int

Table 5.1: Rules for each possible path in figure 5.6 and their corresponding Flow Control Blocks.

circles in figure 5.6, all the necessary information is included in the FCB, which starts with the list of next hops.

Some components need more than just knowing the traffic class of the packet to handle it properly. The load-balancer is stateful and needs an integer per TCP or UDP session to remember which server it chose to handle that session, while the HTTP filter needs some space per TCP session to reconstruct the TCP streams.

We allow each component to describe, on top of the traffic class, the sessions they want to see, and an amount of space they need per-session to write their metadata, the scratchpad. That space will be assigned in the FCB session space, as in table 5.1. To define sessions, rules allow special header wildcards that are *Populate From Packet (PFP)*. The PFP entries mean that the rule must be duplicated with the exact values of the fields when the rule is matched. The difference between defining a rule such as $proto = TCP; dstport = *$ and $proto = TCP; dstport = PFP$ is that the second one will lead to one session and therefore one scratchpad per TCP destination port. When a rule containing a PFP field is matched, the rule is duplicated replacing the PFP with the actual value read in the packet. The FCB will be duplicated along the way, with the needed space ready for each middlebox that asked for it.

Table 5.2 shows an example of the flow table after receiving packets that have hit each of the rules. Only HTTP and UDP rules contain PFP fields, spawning new rules and duplicating the session space. Details of implementation and timeout management are deferred to section 5.6. In the case of TCP packets, as our example only considers the packets with destination port 80 and drops the others, the session mapping will only be done on the last 3 tuples.

Rules							Flow Control Block	
Ethertype	ARP Type	Proto	Dport	Sport	Dst	Src	Next hops	Session space
ARP	Request	*	*	*	*	*	1, 2	-
ARP	Reply	*	*	*	*	*	1, 1	-
IP	*	TCP	80	52100	10.0.0.1	89.18.17.216	2, 2, 1	0x1a234579, 0
IP	*	TCP	80	32100	10.0.0.1	18.17.62.29	2, 2, 1	0x9e5cc632, 1
IP	*	TCP	80	52100	10.0.0.17	120.12.17.12	2, 2, 1	0xab38977d, 0
IP	*	TCP	80	PFP	PFP	PFP	2, 2, 1	TCPSession, int
IP	*	UDP	32100	32100	10.0.0.78	129.251.324.118	2, 1	0
IP	*	UDP	PFP	PFP	PFP	PFP	2, 1	int

Table 5.2: Rules for each possible path in figure 5.6 and their corresponding Flow Control Blocks after having received some packets

5.4.1 Session data size

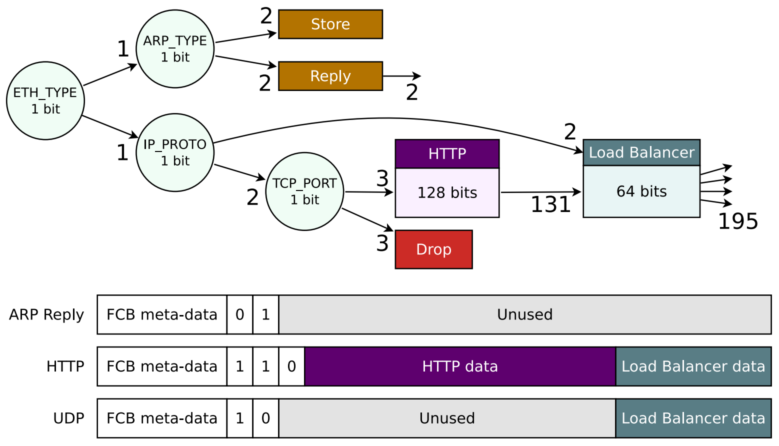


Figure 5.7: Example FCBs showing the computation of the size and offsets needed for each VNFs. Note how the load-balancer data is always at the same offset.

All components of the middlebox must be visited to compute the total FCB size. Starting from each input, the system visits the downstream components and computes the total size needed for all of them. When a packet can traverse parallel paths exclusively, the same space can be assigned to all these paths to optimize space. The components are then informed of an offset in the FCB where they will be able to find their requested space. To avoid needing an indirection table, we prefer to lose some space and have an offset independent of the input path so each element has one and only one offset inside all potential FCBs assigned to packets passing by, eventually leading to some unused space. In figure 5.7 it would seem better to keep the Load Balancer data for UDP at the beginning of the block, but given that FCBs are pool-allocated leading to constant memory allocation costs and that offsets are unique, we have the intuition that a perfect space optimization is not worth it. In terms of cache lines efficiency and data placement, removing the unused space would lead to an unordered access for the HTTP packets as they would jump to the HTTP data and go back to the load balancer data afterwards. It may be possible that re-ordering the component offsets according to flow statistics would slightly improve the performances. But we keep that as future work as it would need dynamic rearrangements and implies much more precautions in multi-threaded environments.

5.4.2 Dynamic scratchpad space and virtualized environments

If the data needed by a middlebox has variable size, the middlebox may simply ask for space to fit its static data and a pointer. The pointer can then be used to keep a reference to memory allocated when the flow is first seen using another dynamic mechanism, such as an efficient pool-allocation system.

This is also the design we propose for a virtualized environment. The flow manager can handle a flow table unique for all VNFs components residing in multiple virtual machines, or more generally, residing in isolated memory spaces. The flow table is kept at the level of the hypervisor as shown in figure 5.8. The hypervisor reserves enough

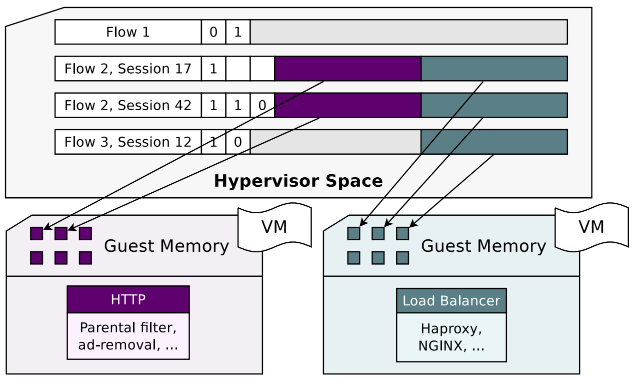


Figure 5.8: Memory allocation for isolated environments

space in the flow control blocks to keep a pointer instead of the session data itself for each VNF. When a “PFP” rule is matched, the flow control block will be duplicated. The hypervisor can then allocate the amount of space needed per-session for each VMs in guest memory when the flow is seen for the first time. This enables fast allocation of per-session space in the guest system without traversing per-VMs flow tables, but still ensuring memory isolation. A practical implementation would need a new API to pass the allocated memory along with the batches of packets from the same session, something still not supported by our prototype.

5.4.3 Multiple levels of sessions

In a middlebox context, it sometimes makes sense to group all traffic according to a first reduced session definition. *E.g.* group packets per IP pairs, check that the pair does not exceed its fair amount of traffic according to some per-protocol and per-endpoints statistics no matter the ports. Then re-classify further the packets with the same IP-pair into TCP sessions. Such a scenario is depicted in figure 5.9, but a practical use case may need even more levels.

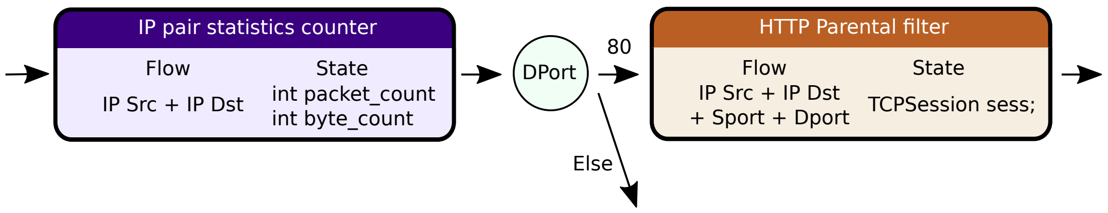


Figure 5.9: Two components needing two different levels of sessions.

In this context, the problem would be that duplicating the FCB for each TCP session for the HTTP Parental filter would also duplicate the space for the IP Pair counters, which would be a problem if the client has multiple different HTTP sessions towards the same server.

When that situation arises, multiple FCBs have to be used. The first one will have the scratchpad space for the IP pair statistics, the second one for the full TCP session. The last field of the first (IP pair) FCB will be used as a pointer to a second flow table, which will lead to the second FCB by matching only the last 2-tuples. The implementation proposed in section 5.6.1 keeps one sub-table per-IP pair, allowing the second table to be very small and only contain TCP sessions for the given IP pair. This will lead to many sub-tables allocations and de-allocations as there will be one per-IP pair. Therefore we use a pool of sub-flow table for efficient memory management.

5.5 Stream abstraction

At this point of our design, a middlebox developer can easily receive a bunch of raw packets matching a given traffic class along with their FCB. The developer knows directly to which kind of traffic belongs the given packets, as this is marked in the FCB. If the component asked for some per-session scratchpad, the middlebox component will also have some space for its own use in the FCB, knowing the FCB has been duplicated per-session and this space is therefore shared by all packets of the same session.

5.5.1 Contexts

But most of the time, a middlebox component developer expects a seamless stream of data, not packets matching a given set of tuples, but from a given protocol. The developer also wants a way to touch the data without caring about the protocol details.

Therefore we introduce the concept of stream context. Middlebox components exchange packets in batches of packets of the same session. When in a given context, components can use a *content offset*, a metadata associated with each packet, to access the payload directly. The context also allows to issue requests to act on or modify the stream. Table 5.3 summarizes the available requests that will be detailed further. On top of these facilities, we offer multiple abstractions that allow to act on the data as a stream, without the need to copy the packet payload, like an iterator that can iterate across packets. This enables zero-copy inspection of a stream, but still allowing the middlebox component using this higher-level stream abstraction to access the headers if need be, a feature all IDS need as some attacks may be based on header fields.

When a middlebox is in IP context, the content offset is set just after the IP header. If the middlebox modifies data in a such a way that the IP packet length changes, the length in the IP header will be changed when the packet leaves the IP context, and the checksum will be updated accordingly. This is showed by the downwards arrow in figure 5.10.

	Packet
insertBytes	Insert some bytes at the given position
removeBytes	Remove bytes at the given position
requestMorePackets	Launch protocol specifics when stalling packets in a buffer *E.g.* TCP should pro-actively ACK the current packet
closeConnection	Close the connection May do nothing if the current context has no notion of connection
registerConnectionClose	Register a function to call when the connection is closing
isLastUsefulPacket	Tell if some packet is the last useful one in the session
determineFlowDirection	For protocol with multiple sides, return an index for the side *E.g.* 0 or 1 for TCP

Table 5.3: Context requests

In TCP context, the offset is moved forward after the TCP header of each packet. Each context does its own work when handling a request and then passes the request to the lower context.

Modification of a packet is a little bit more complex and is shown by the green boxes and lines in figure 5.10. Modification of the number of bytes in a TCP stream implies a lot of accounting around acknowledgement and sequence numbers detailed in section 5.5.5, for the current packet but also the following ones. But still, the IP header will need to be changed no matter what TCP does if the packet length changes, so the request is passed to the lower layer.

Each middlebox component can tell in which context it wants to be. The TCP context will also ensure that the current flow has a session described by its 4-tuple, as the packets only make sense while grouped as such.

On top of the functions to modify the packets, the context also allows to determine if a given packet is the last useful one for the current context. In TCP context, the request will simply check the TCP session state while HTTP context will use the value of *Content-Length* or pass it to the previous context if unknown. The end of an HTTP context does not necessarily mean the end of a TCP context but the contrary is true.

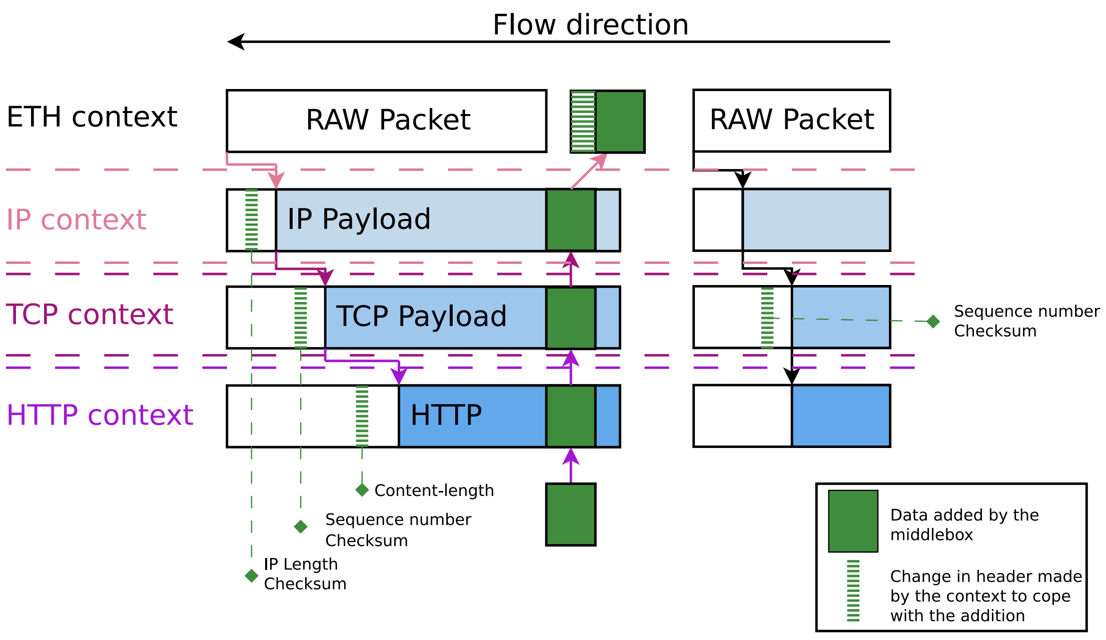


Figure 5.10: Context approach. Upon entry in a context, the payload offset is moved forward. When the stream is modified, previous layers of context take care of the implications, such as changing the TCP ACK number of the current packet and the following ones, or fragmenting the Ethernet frame if need be.

5.5.2 Request for more data

The last request of the context is the ability to wait for more packets. IDS and IPS systems, content filters and many more need such a feature to be able to find patterns across packets. An HTTP ad-remover, or anti-tracker will typically look for known HTML script blocks like $< script > [data] < /script >$ and remove them. If that script spans across multiple packets, the second part of the HTML block could arrive not right after the first part, therefore the system must provide an efficient way to wait for more packets.

For example, a `TCPReorder` element that will re-arrange TCP packets so they get out only in order needs space in the per-TCP session scratchpad for a “out of order packet pointer” and a “last seen sequence number”. When an out-of-order packet arrives, `TCPReorder` sets those pointers and returns. In the run-to-completion model, the CPU goes back to the flow manager. When a packet of the same session arrives later on, it will be pushed up to the `TCPReorder` that will check if there are some “waiting packets” in the scratchpad, reorder them, and sends all in-order packets if the hole has been filled.

The goal of our work is not to review ways to handle missing packets, or different IDS/IPS implementations. Our platform offers some services that may or may not be used by more specific implementations to build upon with the factorization of the classification and “only-done-once” session classification advantage.

5.5.3 TCP flow reordering

The TCP context will also take care of re-ordering packets for VNFs that need to receive a stream of ordered payload. The TCP context will send pro-active ACKs when a hole is encountered and starts buffering packets that are out-of-order and therefore cannot be passed yet to the VNFs.

Functions that does not need to see a stream of data such as NATs or load-balancers can receive out-of-order packets. The TCP context entry will not reorder packets for those functions.

5.5.4 TCP flow stalling

As a TCP source may wait for an ACK from the destination before sending more packets, buffering data may prevent the destination from sending an ACK, leading to a deadlock situation. The TCP context provides an optional functionality to do pro-active ACKing.

If enabled, when it receives a “request for more packet”, the TCP context sends an ACK corresponding to the given packet to the source with an acknowledgement number corresponding to the sequence that the destination would have send. We therefore keep a “shallow copy”[1] of outgoing pre-ACKed TCP packets in a buffer until the destination acknowledge them. Buffering is done when a middlebox component specifies that it may stall or modify packets, or when the component wants to protect against TCP

[1]We do not copy the packet content, we use buffer reference counting. The packets leaving the TCP context will be remembered (using a linked list of pointers) and have their usage counters incremented by one, most likely to 2. When sent, the packets will have their usage counter decremented, most likely to 1. When the ACK is received, the list will be pruned and the packet’s counters decremented again, and the packet recycled when the reference counter reaches zero.

overlapping segment attacks[1]. If no VNFs in the service chain need to see an ordered stream of data and can process the same data twice, we do not need to keep outgoing packets in buffers, as processing retransmissions does not pose any problems, those are actually the same VNFs that the one which do not need reordering.

Another more traditional approach also supported by our platform when the flow is not to be modified such as for analysis purposes is to let the packets go through even if it may be part of malicious content. The ACK is sent by the destination as expected, and we send a RST to both sides of the connection if the connection needs to be closed when a further packet is received, *e.g.* when a pattern has been matched by an IDS. However some protocols on top of TCP may have already handled the payload and most of the attack executed, or content unfiltered could have been displayed. On the other hand, a valid example for such approach is the case of HTTP file downloads, where *in general* the file will be dropped if the connection is reset before the last packet is received.

5.5.5 TCP flow resizing

Stalling and re-ordering are requirements for modifying a stream. Many applications need to modify the stream content. For the specific web case, examples include rewriting HTTP traffic to change URLs per CDN-based ones, ad-insertion and removal, along with potential new uses enabled by the novel performance of the lightweight in-the-middle stack we propose such as per-user targeted HTTP page modification or a proxy cache that would include image content in the page itself. Instead of dropping a flow containing an attack it could be sanitized to keep only the original content. Pages could be translated on the fly to target the user language. Other usage include some protocol translator, video transcoding or audio enhancement. As will be shown in the evaluation section, only a few lines are needed to build innovative uses on top of our framework.

The layered context approach allows to propagate the effect of modifications to lower layers. When the middlebox removes or adds data in a TCP stream, the sequence number must be set accordingly so the destination does not think the data has been

[1]A valid segment of data pass through the IDS, then the attacker sends a retransmission for that data with a different content that would not go through the IDS pattern matcher to prevent messing the state of the pattern matcher and would be directly passed to the destination. Depending on the implementation, the destination may keep the malicious data instead of the first received segment.

lost or is a duplicate. However when the destination sends the corresponding ACK, the number must be mapped back to its original value.

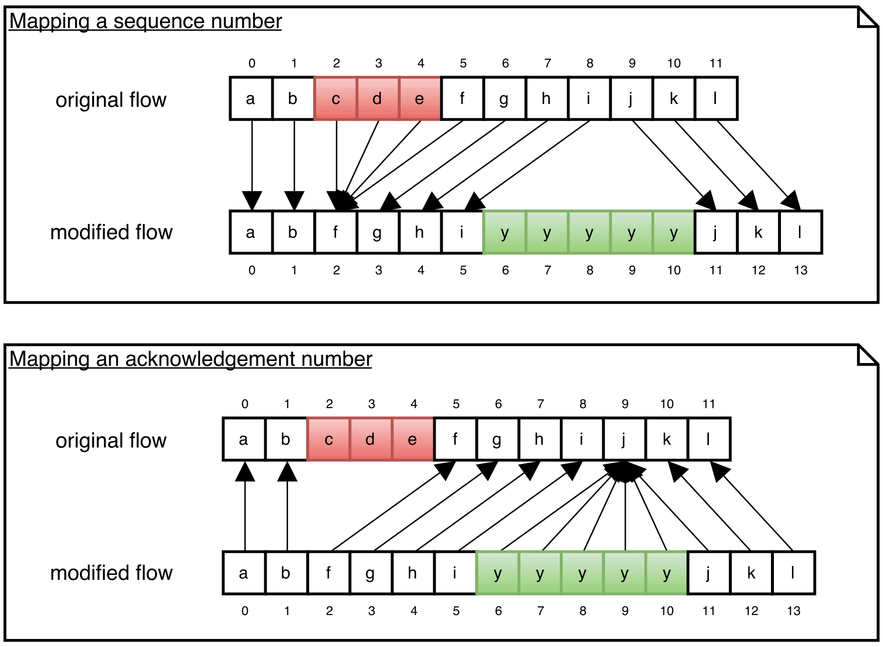


Figure 5.11: Example of the mapping between an original flow and the corresponding modified flow. Red cells correspond to data removed from the original flow and green cells correspond to data added in the modified flow.

The figure 5.11 shows an example in which the following modifications are done to the original flow by the components in the TCP context, or above:

A. 3 bytes are removed at the position 2, removing *cde* from the stream.

B. 5 bytes are inserted at the position 6, adding *yyyyy* in the stream.

An important point depicted on figure 5.11 is that, to map a sequence number, the positions corresponding to the removed bytes are all mapped to the position 2, corresponding to *f*. Indeed, if the sender starts a retransmission with a sequence number equals to, for instance, 3, trying to retransmit data from *d*, the mapped retransmission will start at *f*. On the other hand, when mapping an acknowledgement number, all the added data point to *j* in the original flow.

We keep track of a list of modifications, represented by a position (the position at which the modification occurred) and an offset that corresponds to the number of bytes modified. This offset is negative if bytes are removed and positive if bytes are added. We cannot keep a cumulative offset because an ACK may come and ask for previous data before the cumulative offset and we would not know which bytes have been removed or added from the original flow. However the list can be pruned when an ACK arrives.

It is worth mentioning that the book-keeping system will only be instantiated and used if one of the middlebox component specify it may resize some flow. When the size of the flow is not going to change, there is no need for such sequence mapping.

When resizing is enabled, the system must also keep the modified data in a buffer for potential retransmission. The data to be maintained is limited to the currently un-ACKed modified data, which in general represents less memory than a usual end-point TCP socket. The buffering is done when leaving the TCP context, that is in most cases after all potential re-writing.

5.5.6 Matching both directions of the flow

Some data must be shared between both directions of a session, such as TCP sessions. Parsing again the classification tree but with inverted destination and port could have been an option. But if both directions are not ensured to be served by the same core, the tree would need complex locking solutions which would slow down probably the most important element in the fast path. Ensuring that both directions are served per the same core, is not as easy as it seems.

A symmetric hash key[102] could be used, but if NAT is in place the return direction will have a different tuple that will not hash to the same core. mOS[46] loop through multiple NAT source ports and compute the 4-tuple hash in software in the same way the hardware would do until the hash leads to the current core as RSS would. This approach is theoretically unbounded, though in practice the number of tuples to hash in software should be equal to the number of cores. Still on average as many hash as cores have to be computed in software.

Using the capabilities of smart NICs may allow to allocate NAT ports in such a way that the return packets will be served by the right core, using specific receive filters to assign chosen ranges of ports to core. The most commonly seen Intel NICs in recent research papers, 10G 82599-based ones, had some unique field masking support but it was dropped by Intel in its later XL710 chipsets which is beginning to replace 82599-based ones as the new generation of chipsets and because it supports 40G. While there are a lot of Smart NICs with more possibilities[103, 104] we would have to support a fall-back solution for less-capable hardware anyway. NICs also often lack support for newer protocols when they arrive, for example, IPv6 port filtering is still not supported in the XL710 at the time of writing.

Therefore we prefer to develop an efficient, lockless (as much as possible) solution that will allow each side of the connection to be served by different cores, but proposes a very efficient way to reconcile common data.

When a new TCP stream is seen (a SYN packet), the TCP context entry will allocate a new common data structure for data common to both directions using a per- thread memory pool. The pointer to the common data is saved in the session scratchpad of the FCB. The TCP context entry component will add a pointer to this common space in a thread-safe hash table where each bucket is protected by a *Readers-Writer (RW)* lock. The RW lock is based on a usage counter. The counter will be negative while there is one writer and positive when there are readers in the bucket list.

When the other side sees the corresponding stream, it looks for the inverted 4-tuples in the hash table, adds the pointer to the common data to his session scratchpad. As the session scratchpad in the FCB will be passed with each packet of the same session, the hash table will never be read anymore for the same session and the entry from the hash table can be removed.

Section 4.3 discusses implementations for efficient concurrent access to the data structures in the common area. We found that in most cases, no locking is needed at all. The most common operation in the TCP case is to look at the other side's current ACK number. Only the other side will write the ACK number, and as x86 as a coherent cache hierarchy, the ACK number can directly be read. Updating the TCP state does not need to be atomic either. Only one side will send a SYN, then the other side will send the SYN ACK. The same goes for the closing handshake. In the very unlikely case that both side send a FIN on their own and they are processed at the very same moment on both cores, potential consequences are only one side being closed too early. The only plain old lock needed is around the sequence maintenance structure needed to allow flow resizing. Further research could use the fact that an ACK only relates to data that passed through previously to develop a specific data-structure that even completely avoid locking.

5.6 Prototype implementation

In the search for an efficient middlebox platform on commodity hardware, we established some criteria:

- Flexible *module-oriented* configuration to allow users to easily use our system.
- Possibility of hardware offloading of some functions.
- Be in user-space to facilitate communication with other user-space software in a zero-copy fashion using shared memory, and ease the development compared to kernel programming.
- Support for fast packet I/O engine (Netmap[32], PSIO[35], PF_RING[33], DPDK [31], *etc.*).
- Common networking functions already implemented, to reduce development time.
- Stateful packet processing, and more generally support for flow reconstruction for flow treatment like DPI functions, HTTP reconstructions, and other middlebox features.

Chapter 3 built such a platform, FastClick, which implements all criteria but the last. Therefore, we implemented the flow design as described in the previous section to make Click elements match the component definition given in section 5.2, mainly being able to act on streams instead of packets, and asking for some flows and per-session scratchpad by allowing them to override a few virtual functions. We'll reference Click in the following sections for common Click functions, only mentioning FastClick when its specific features are involved.

In Click, the service chain is defined as a set of elements piped together. Dispatching of the traffic according to fields, the flow classification, is done using a `Classifier` element (or its variants such as `IPClassifier` that provide more convenience) which dispatches traffic to following elements according to a given set of rules, as shown in figure 5.12 (a).

In our flow-aware FastClick extension, which we call *MiddleClick*, we introduce a `FlowClassifier` that must be put just after all `FromDevice` elements, that will visit all downstream elements to combine all flow classifications and build a classification tree.

```
ct :: Classifier(12/0800,
                 12/0806 20/0001,
                 12/0806 20/0002);
cp :: IPClassifier(proto tcp, proto udp, -);
cd :: IPClassifier(dst tcp port 80, -);
td :: ToDevice(...)
arp_querier :: ARPQuerier(...) -> td;
lb :: LoadBalancer() -> arp_querier;
FromDevice(...) -> c0;
ct[0] -> cp;
ct[1] -> ARPResponder(...)[0];
ct[2] -> [1]arp_querier;
cp[0] -> cd;
cp[1] -> lb;
cd[0] -> HTTPProcessor() -> lb;
```

(a)

```
td ::ToDevice(...);
arp_querier :: ARPQuerier(...) -> td;
lb :: LoadBalancer() -> arp_querier;
fc :: FlowClassifier();
fd :: FromDevice(...) -> fc;
fd ~> ARPResponder(...)[0] -> td;
fd ~> [1]arp_querier;
fd ~> HTTPProcessor() -> lb;
fd ~> lb;
```

(b)

Figure 5.12: (a) Click configuration for the example of figure 5.6. (b) Corresponding MiddleClick configuration.

The difference with the usual Click classification is that the rules are aggregated before Click enters the running phase. `FlowClassifiers` initialize the session scratchpads with the next hop numbers (the output port numbers in Click). Traditional Click classification elements such as `Classifiers` are replaced by `FlowDispatcher` elements which follow the same syntax. `FlowDispatchers` just have to read their scratchpad in the FCB to decide the output, without classifying in place or even touching the packet. Their only job at run time is to dispatch packets to the right output port according to the next-hop field, hence their name.

Alternatively, figure 5.12 (b) illustrates a new link syntax called the context link, "∼>", which will automatically insert a `FlowDispatcher` element between its two ends. Those `FlowDispatcher` elements are configured according to flow descriptions exported by all elements to the right of the arrow. Context also allows to remove the needs for obvious `Classifier`, in our example the input can directly be tied using the context link to all ARP elements, the traffic class defined by the ARP elements will be used through the inserted FlowDispatcher to actually give ARP requests to the ARPResponder, replies to the ARPQuerier and other packets to the remaining path. In many cases, the element will always ask for the same traffic class and an explicit `FlowDispatcher` is not needed.

To specify a session, the system needs an offset and a mask. The mask will tell which bits define the session, that is the PFP fields. All packets sharing the same masked value at the specified offset will share the same session. The definition for a TCP session would be `ipsrc/ffffffff ipdst/ffffffff dport/ffff sport/ffff`. If the port was already known, as in our running example, the `dport/ffff` will be ignored when the rule is merged by the `FlowClassifier` when doing the graph traversal when Click launches, avoiding to reclassify on the same tuple. All elements placed after such flow dispatchers that define a session rules will have a session scratchpad per TCP session. Using the context link, TCP and UDP specific elements will also automatically insert such a session definition, in practice it is rarely needed to be written down by the operator.

The flow and session definitions are totally flexible. *E.g.* a component can ask to receive only packets matching an arbitrary value at an arbitrary offset, or can define arbitrary PFP fields to define a session that is completely different from the TCP 4-tuples. The implementation of the software classification algorithm inside the `FromClassifier` that is subject of section 5.6.1 actually allows for programmable classification instruction that return an arbitrary value when given a packet. Rules such as *dport/ffff* translate into a "$16bit$ header field at offset $transport + 2$" that given a packet will return its destination port. Nothing prevents more *evolved fields* that would return a number according to some HTTP cookies in the packet or run a BPF program on the packet that returns some values. The only limit is that the advanced field cannot modify the packet, as the classification is the very first step and supposed to be fast and read only. Introducing unknown fields also prevents hardware offloading, though the advance in recent NICs allow them to run complex BPF programs or P4[105] enabled upfront hardware could run more complex functions, tag the packets, and let the *evolved field* read the tag.

FastClick already implements batching using linked lists to pass lists of packets between elements, instead of single packets. In MiddleClick, batches of packets are always packets of the same session. To avoid having small batches, the flow classifier has a builder mode to put packet in an internal ring of batches. The FromDevice element passes a batch of packets to the FlowClassifier. The FlowClassifier then classify the packet and insert it into the ring. The FlowClassifier then classifies the next packet, and searches the ring for packets of the same session (that lead to the same FCB), and appends the packet to the end of the batch if found. When all packets are classified, it sends the session batches to the next element one by one. If the ring is full but the packet that was just classifies is not from the same session than any batch in the ring, the oldest batch in the ring is processed directly. Packets are reordered but the relative order inside the same session is kept, and as the array is processed in order, the relative order in which sessions are seen is also kept.

Hence, any middlebox element knows that it can work on the payload of the packets of a batch as a single stream of data. Instead of modifying all push() functions of Click to pass the FCB along the packet, we preferred to use a thread-local global variable that is set by the FromClassifier before pushing the batch through the graph. Each element can access from any function. In a run to completion model the FCB pointer is guaranteed to be the one of the current batch.

5.6.1 Flow Classification

A simple implementation would be to have a static classification algorithm that leads to either FCB pointers or PFP rules. If the classification leads to a FCB, it is used as is. When encountering a PFP rule, a fast dynamic classification algorithm can be used to search for the rule with the PFP fields expanded. This is in some way what traditional Operating Systems do for TCP. Some fixed-at-compile-time lookup is done on the packet headers (ethernet type, ip protocol, ip version, ...). This part is the static classification, as it is not subject to change. If the packet is a TCP packet, then the 4 tuples will be matched using a hash-table. UDP and other protocols would use another hashtable.

However the traditional approach is not generic enough. The static classification must be flexible, allow for non-standard field-based classification and built according to the VNFs in the service chain. The dynamic fields used in a hashtable-like data structure

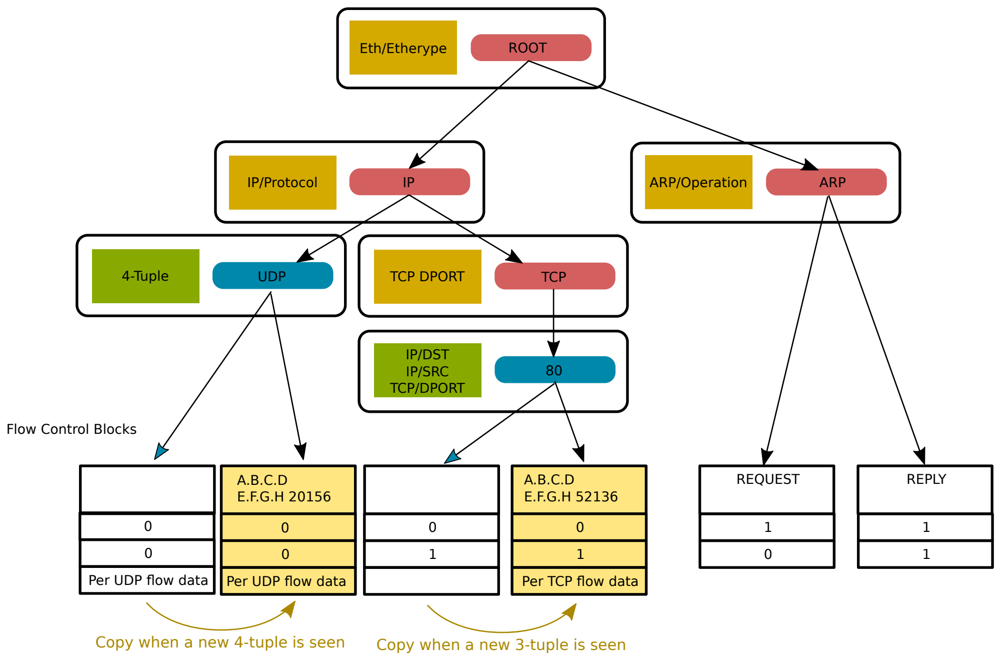


Figure 5.13: Classification tree for the small setup of figure 5.6.

must also be flexible, and the fact that some fields will be known when reached from a specific path and therefore do not need to be used in the dynamic classification should be taken into account to allow improvements.

In our implementation, traffic classes and session classification are both implemented using the same tree. Each node of the tree corresponds to one field, where each node may have its own classification implementation according to the associated field or user-provided hints about the best underlying implementation. A condition if there are two possible outcomes, an array if the range of possible values is not big (such as with VLAN numbers or 1-byte fields), a heap for static classification or a hash-table in the other cases. Therefore the term "tree" does not refer to a usual algorithmic tree but a more flexible implementation. Figure 5.13 shows the classifier corresponding to our running example.

In our example, the first three nodes of the tree will use a condition as only very few values are possible.

Each node also links to a level object (square objects in figure 5.13) that defines the programmable field implementation, *i.e.* how to extract data from the packet for header fields, or more complex logic for more evolved fields. Most levels are header classifiers that will read a field of the packet, apply a mask and return the value so the node implementation can access a child according to the value.

Matching simply consists in descending the tree until a leaf is reached. Leaves are not node objects, but directly FCBs. All nodes also have a default branch. When building the tree, we ensure that all paths (children branches or default branches) actually lead to some FCB. In the example of figure 5.13, an ARP packet that is not a request or a reply would go to the default FCB of the ARP level that will have a special "next hop" that the FlowDispatcher will recognised, and will drop the packet. FCB can be marked to apply early drop directly in the FlowClassifier if the operator allows it. It is not always desirable because the operator may want to keep statistics about packets being dropped. Level objects also define if the field is a PFP field in which case the default node is to be duplicated upon a miss. When the default node is duplicated, it will also duplicate the child FCB, therefore leading to a per-session FCB.

FCBs are managed in per-thread pools for efficient allocation and recycling.

It is important for the reader to understand that the classification algorithm we propose in this section is only one example of how to implement a flow table that allows dynamic duplication of some rule. We could have used a very efficient classification algorithm like HyperCuts[106], but its update rate is far too slow. In our tree implementation, each node may use different implementation according to its particularities (static, dynamic, maximal number of values, ...). Using more tailored data structures than the basic ones we implemented is left as future work.

If the Flow Classifier is to be traversed by multiple threads (detected using FastClick's thread vector), a special dynamic "thread" node will be inserted before the first dynamic node. When a new thread passes through the thread node, it will duplicate its children for the current thread. That is correct under the assumption that packets for the same session are handled by the same core, a feature allowed by RSS hashing. This should not be confused with packets of both side of a stream that can be handled by different cores and will therefore use different branches of the tree as they will branch differently when encountering the thread node. Section 5.5.6 already explained our approach to reconcile the two sides.

5.6.2 Classification tree expansion

When the default implementation of a node is not sufficient (*e.g.* it reaches its maximal capability, or it is producing a lot of collisions for a hashtable, ...) the node starts to grow, which is marked by a flag in the node. The default path of the node is replaced by a new empty node that uses the same level but a more appropriate implementation, like a bigger hashtable, or when the hashtable size is closing up to the amount of possible values (like the 65536 possible values of the 16bits TCP/UDP ports fields), a vector with one child for every possible values. When the node is growing, no more children can be added, the classification only looks at the node for existing flow When all children of a growing node are removed (flow timed out or finished), the growing node is removed and only the bigger replacement node remains. This scheme allows to avoid jitter caused by growing hashtables that normally needs a full re-allocation. It would also allows to grow one badly performing implementation into another one, *e.g.* changing the hash function of a hash table, moving from linear probing to open addressing, etc.

5.6.3 FCB release

Each FCB has a usage counter. When a packet matches a FCB, the FCB usage counter is incremented, and when the packet is released, the usage counter is decremented. When the FCB usage counter reaches zero, a user-defined function is called to release some FCB state that may need to be clean. The FCB is put back in the pool but also removed from the tree. All nodes and leaves (FCBs) have a parent pointer so all dynamic (PFP-duplicated) parent nodes having no other children can also be removed to prune the tree.

That means that if any middlebox component wants to keep the session open longer than for a burst of packets, it must increment the usage counter to take a reference and release it when it sees the end of the stream, such as a RST or a FIN flag for TCP sessions.

As an alternative solution, the flow manager also implements a global timer. To avoid managing timeout themselves, middleboxes do not take any reference on the flow, but set an amount of time the flow will stay alive even if the usage counter reaches zero. Middleboxes are encouraged to use the global system to keep a consistent state across middleboxes. If a NAT decides to drop a flow, we should also release it in subsequent middleboxes as the flow will be broken anyway. Releasing is done directly when the usage counter reaches 0 and the timeout is passed. If the timeout is not passed, the FCB will be added to a list of pending timeouts.

The question is thus, when to look through the list of pending timeouts for expired sessions.

In Click, tasks return false if they did not do anything useful. This was used for scheduling purpose. We added a special kind of Click task, called `IdleTask`. An `IdleTask` will run when all other tasks returns false, meaning that the thread is idle. We use an IdleTask to trigger the check for expired timeouts. As an IdleTask could technically never run, we also look for expired timeouts if the list of pending timeouts is longer than some threshold after matching a full burst of packets. The threshold uses exponential back-off as the fact that the list of unexpired FCBs growing is normal. To provide a final upper bound on the release time, a timer also runs the list pruning every 15 seconds. Hence middleboxes needing precise timeouts must manage their own timing as the list might not be released fast enough.

If a FCB is matched while in the list of pending timeout, it is checked for expiration right away. If it is expires, the FCB will be renewed by the classifier as if it had expired (the release function is called and the FCB space resets to its initial value), ensuring that even if release is delayed up to 15 seconds, the timeout time itself is more strict.

Implementation details: *the FlowElement classes*

Any element that is part of the flow system (either defines a traffic class or wants some per-session scratchpad in the FCB) extends the $FlowElement$ class instead of FastClick's $BatchElement$ class. $FlowElement$ has a few virtual functions that can or must be implemented to define the traffic class, the amount of space the element wants into the FCB and its initial value. $FlowElement$ also have a _flow_data_offset integer that will be set by the $FlowClassifier$ to the offset in the FCB where the scratchpad space is to be found. Note that virtual functions are only called at initialization time. At run-time the inherited element can look for its space at $fcb\_stack->data[\_flow\_data\_offset]$.

For convenience, a template $FlowSpaceElement < Derived, T >$ is provided. It will implement the virtual $FlowElement$ functions that return the per-FCB state for a given structure T. The operator will have to implement the *push_batch(int port, T* fcb, PacketBatch* batch)* function instead of FastClick's usual *push_batch(int port, PacketBatch* batch)* function. The new $T * fcb$ passed will be a pointer to the scratchpad space in the FCB, computed as explained above. The Derived template parameter is used to rely on CRTP instead of a virtual function call to implement *push_batch*. In itself, FlowSpaceElement is only a few lines, returning $sizeof(T)$ when asked for the amount of space it wants, and passing *fcb_stack->data[_flow_data_offset]* to Derived's *push_batch* function.

A deeper $FlowStateElement < Derived, T >$ class is provided for elements that wants to build upon an event-driven idiom. It builds on top of FlowSpaceElement, but allow the user to define 2 new functions: *bool new_flow(T*, Packet*)* called when a new flow is seen for the first time, *release_flow(T*)* called when a flow is dying, and a time-out value to force the flow to stay alive for some time. The T structure is enclosed with a boolean to remember if the flow has been seen. The timeout is managed by the flow manager and a wrapper around release_flow is registered in the FCB's release function chain.

The template approach may seem a bit unusual when thinking of event driven systems, where most frameworks allows to register functions to be called when some events happen. Our approach allows to avoid potentially long lists of function to call for each event. In our approach only the release is part of a traditional function list, though if the element implements its own timer it can also be avoided. The CRTP pattern actually avoid any function call on top of FastClick's push_batch, leading to a very efficient implementation.

■

5.6.4 Context implementation

Entry and exit of context are done through pairs of IN and OUT elements, such as `TCPIn` and `TCPOut`.

IN elements traverse the graph to find their corresponding OUT elements and announce themselves to them so one can access the other. Along the way, they also announce themselves to the next flow elements, including other "downstream" IN elements. A middlebox element will have a *previous context* pointer pointing to the last IN element, which will have its own pointer to the previous one, *etc.* up to the first IN element.

Each context entry element implements a set of known requests described in section 5.5.1 (packet is modified in a certain range, request for stalling, know if the packet is the last one of a flow-based context, ...). For each request, the context entry element will execute its protocol specifics, and then pass the request to the previous one and so on, until the first entry element (IPIn in most cases) finds no other context entry. Combined with the context link, the usually complex Click manual wiring is actually very minimal as shown in figure 5.14. When the flow classifier traverse the graph and resolves the $\sim>$ context links, it remembers the last IN element that was traversed.

The context link already allows to spawn a classification rule for a traffic class and session definition according to the elements on the right of the $\sim>$ symbol. We also wanted to implement a way to automatically define protocol classification when IN elements are inserted. For instance, the IPIn element could not expose a traffic rule such as *ethertype 0800* because the IP header may be encapsulated over another protocol, *e.g.* over a GTP tunnel, and not in an Ethernet frame. Therefore the last IN element is interrogated to spawn a traffic class rule according the next IN element. This allows the IPIn element to spawn a *ip prot/06* traffic class rule when it is followed by a TCPIn element. But if TCPIn was preceded by an ATMIn element to implement TCP-over-ATM, a different traffic class rule would be used if ATMIn has been programmed to return a rule when interrogated about how to classify for TCPIn. By default, the `FlowClassifier` act as a first "EthernetIn", returning a rule like *ethertype/0800* when a context link is inserted before an IPIn element. Therefore the example of figure 5.14 will actually have a flow table with one rule "etherype/0800 ip prot/06 ip src/fffffff ip dst/fffffff udp src port/ffff udp dst port/ffff" that will drop non-IP/UDP packets and duplicate itself along with the session scratchpad for each UDP 4-tuples.

In this way, we keep Click's modularity but have a very much streamlined default case. Context links can be omitted to use a more refined `FlowDispatcher` if the user wants a finer control on classification.

```
FromDevice(...) -> FlowClassifier ~> IPIn ~> UDPIn(TIMEOUT 300) ~>
 WordMatcher(ATTACK, MODE REMOVE) -> UDPOut -> IPOut -> ToDevice(1);
```

Figure 5.14: Configuration for a transparent middlebox that removes the word "ATTACK" of UDP flow passing by, even across packets. As UDP does not implement connection semantics, the UDPIn element can set the session timeout to some value, here 300 seconds.

Using a HTTPIn context entry after a UDPIn context entry would work out of the box, but of course would behave unpredictably when some packets are lost or reordered, as UDP does not ensure any of those.

5.6.5 Socket-like abstraction

As the elements of a FastClick configuration manipulate batches of packets, it is not convenient for a developer to perform some operations such as searching a specific pattern in the flow. We provide an iterator-like object reminiscent of FlowOS[44]. Initially, the iterator points to the first byte of the current level in the batch of packets. When calling `iterator++`, it may cross a border seamlessly according to the current context. If the processing function returns with the iterator in the middle of a packet, all previous packets before the iterator point will be processed through the rest of the graph. A request for more data will be propagated through the context for the packets after the iterator point. If the iterator was at the last packet, all packets will go through.

This abstraction allows to implement new VNF components that would appear very complex in only a few lines. As with vanilla Click and later FastClick, we expect the base of elements that take advantage of the context/session scratchpad to grow. We already provide multiple generic VNF elements that act on the current context such as regular expression matcher, packet counter, load balancer and an accelerated NAT.

Implementation details: *the StackElement classes*

StackStateElement<Derived, T> is similar to the *FlowStateElement<Derived, T>* except that the event it provides rely on the context and not on arbitrary time-outs and release functions. If *new_flow* returns false, the context's close connection request will be called to kill the flow and discard all subsequent packets. The *release_flow* function will be called when the stack decides to release the flow (such as TCP timing out or receiving a RST or an ACK for a FIN).

StackBufferedElement<Derived, T> is the most refined template implementing the socket-like abstraction explained in this section. Instead of the *push_ batch* function the user needs to implement a *int process_ data(T*, FlowBufferContentIter&)* function that will be called with the iterator over new available data, starting where the user left it at last call. If the function returns a non-zero value, the data is destroyed and the close connection request propagated to the context. If the iterator is returned, but not at the end of the flow, a request for more data is made to the context, and packets before the iterator point are pushed to the next element.

Each of the StackElement and FlowElement helpers only add a few lines over each other, proving the case for CRTP's inlining instead of using register_event functions that call each other leading to long chains of functions doing not much work.

■

5.7 Performance evaluation

This section discusses our results under various test cases. We start by demonstrating the performances in a few empirical or innovative test cases and then study the advantages while building a service chain composed of multiple of those test cases.

5.7.1 Stateless firewall

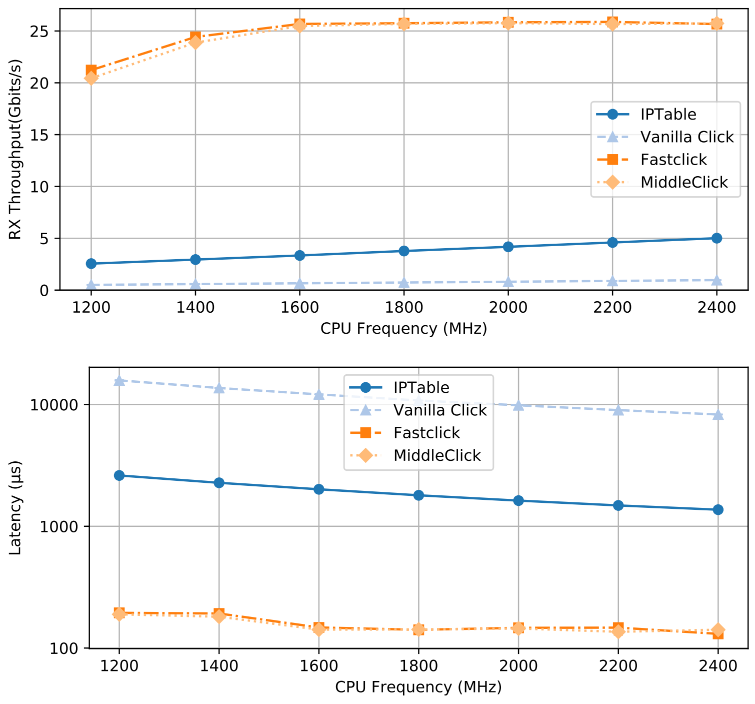


Figure 5.15: WAN to LAN throughput and latency of a stateless firewall using 2 cores (one per side), except for Vanilla Click which uses 4.

To study the impact of the classification process, we built a stateless firewall test case both with iptables, Vanilla Click (using PCAP for I/O), FastClick, and MiddleClick. The *device under test (DUT)* has an Intel® Xeon® E5-2630 v3 CPU with 8 cores at 2.4 GHz, 32 GB of RAM, and is connected to a switch using two 40 Gbps Intel® XL710 cards. The setup is similar to the campus router presented in section 3.2.1, using a second identical *generator* machine to replay traffic traces. However, in this test the *DUT* acts as a firewall between the WAN and the LAN, in a similar position than the one used to capture the traces. The traces are replayed according to their original direction, and the rules are created in consequence (*e.g.* dropping traffic with a source IP address of the internal network coming towards the LAN, accept only known ports, etc) to a list of around 20 rules. This test case is representative of small and mid-end customers who want to do some basic security checks at the first point of connection of their network before sending the traffic towards diverse appliances. The inbound rules drop about 30% of the traffic, while the outbound rules never drop traffic, as our campus has already a firewall that dropped bad outbound traffic before the capture point. 30% of the traffic may seem huge, but the point of capture is actually before the University firewall and it is therefore capturing inbound bad traffic. Also, as this first case is stateless, some inbound packets are rejected as they would usually be accepted because they are related to a connection initiated inside the campus. To ensure consistency (and not especially best performances) we disabled all motherboard turbo abilities, CPU P-states and down or up-scaling. CPU Cores 1 to 4 and corresponding hyperthreads are isolated using the isolcpus Linux command line. The tuning is done according to what was discussed in chapter 3 to max out the performances regarding I/O, batching, ...

To be able to make comparisons, all implementations of the firewall run on two cores, except the Vanilla Click one which uses 4 cores to avoid the receive livelock problem, assigning 2 cores for interrupts and 2 others for Click itself (see chapter 2).

All solutions implement a very simple routing table of two rules to direct the traffic against the right port. This is done to compare more honestly towards the Linux iptables module where routing cannot be avoided. Although technically the firewall could be fully transparent.

Figure 5.15 shows the results of the firewall test case for the WAN to LAN side. We omit the other direction as it is not under high load. As expected, the heavy Linux network stack is falling way behind FastClick or MiddleClick solutions, and Click is even worse as it does just more work after the packet has traversed the OS stack. Again, refer to chapter 2 to understand the problem behind the Vanilla Click configuration.

This test only pushes the MiddleClick classification upfront in a more complex structure which is not completely used as this example is stateless and therefore does not require any session. The test case shows that this much more complex classification structure than the one used by the Click IPFilter element used for the Click and FastClick implementations of the firewall does impact the performances, but it stays minimal.

Moreover, MiddleClick passes packets of the same traffic class and same session between elements, which causes MiddleClick to pass smaller batches between elements, impacting the throughput a little.

Using a single core would be enough to reach the I/O throughput of the traffic replayer, hence we scale the CPU frequency to be able to compare solutions under CPU constraint.

Latency results also show that the addition of the flow classifier does not impact MiddleClick, with an average latency of 160 μs, while iptables has an average latency of 1200 μs. Latency is measured using the same system than section 3.2.1. Latency of packets dropped is omitted, as we have a dropping firewall in the scheme.

5.7.2 NAT

While OpenBox and E2[43] are probably the work closest to MiddleClick, their implementation is not fully available and this prevents comparing the most interesting parts of our work. The per-flow metadata of OpenBox is only conceptual, and they do not tell how they would build and manage a data structure like the FCBs to handle millions of flows per second, which is, in fact most of this chapter. Moreover OpenBox uses the very slow Kernel I/O. Bringing DPDK support to OpenBox would lead to actually re-building FastClick as it is based on Click. E2 is only released as SoftNIC/BESS[107] is released. E2 allows to use "bytestream vports" allowing to avoid TCP reconstruction, and their per-session *Metadata Tags* are not available either. Therefore we would only be able to compare packet based functions, which is what FastClick proposes and is not much of interest in this chapter. mOS[46] on the contrary is fully available and proposes a NAT implementation that we can compare against.

We simulate 128 concurrent HTTP clients using WRK[108] on the *generator* machine that generates requests towards a NAT that will forward the requests towards an HTTP *server* running NGINX. Note that the 128 concurrent clients executes requests in loop, leading to a theorical $\sim 3M$ requests per seconds for 1K files at 40Gbits/s. The *server* is connected to the switch with a 40G Intel® XL710 NIC machine that has an Intel® Xeon® E5-2683 v4 with 16 cores at 2.10GHz. Figure 5.16 shows the performance of a MiddleClick-implemented NAT, the mOS one and the standard Linux NAT.

MiddleClick performs better than both Linux and mOS. mOS also impose serious limitations as its model prevents using L2/L3 features such as learning bridge or ARP

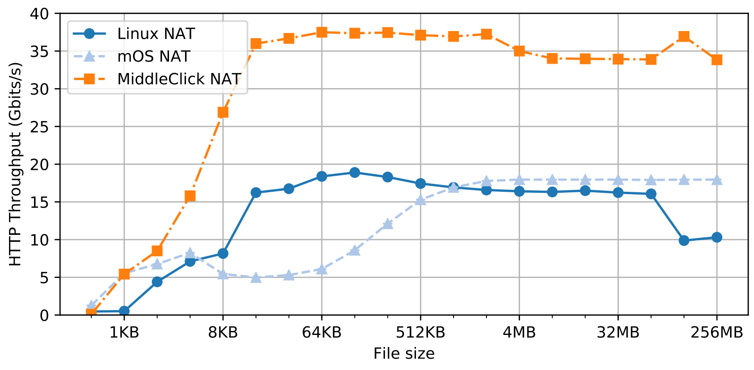


Figure 5.16: Throughput of data downloaded through different NAT implementations using 128 concurrent connections on one core

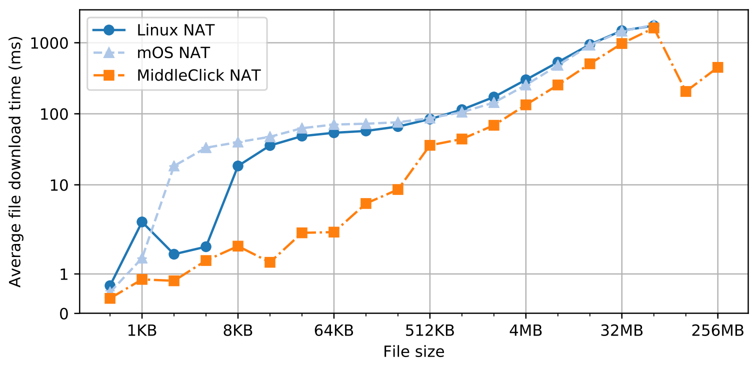


Figure 5.17: Throughput of data downloaded through different NAT implementations using 128 concurrent connections on one core

queries and therefore only supports a bump-in-the-wire configuration when used in inline[1] mode (mandatory for a NAT), needing multiple static ARP configuration on all systems on both sides of the "bumped" wire. In its current state, mOS cannot be used either in a service chain when using DPDK, which is mandatory to have acceptable userlevel performances. The slowness of mOS can be explained by the much heavier TCP stack it brings up, mostly unnecessary for a NAT. mOS reproduce the state of both sides of the connection, speeding therefore a lot of time in timeout management. In our system, we have a unique connection state with a negligible timeout management

[1]inline mode refers to a configuration where packets pass through the box, as opposed to a monitoring mode where the box receives copies of packets to perform some analysis but does not let them through

cost and when the stream is known not to be modified by the NFVs components, the checksums are not fully re-computed. Correcting the TCP checksum after address translation is done directly by computing the difference between the new and the old parameters.

One advantage of mOS is its relative simplicity on the programming side. As discussed in section 5.5.1, we propose a mostly similar event-driven layer, while being a Click extension we still support the modularity prohibited by the TCP-first mOS approach, allowing ARP handling, DHCP support, or TCP over tunnels.

5.7.3 TCP load-balancing reverse proxy

To evaluate the flow performances of our system against state-of-the art industrial solutions, we built a TCP load-balancing reverse proxy. The proxy balances in a round-robin way the upcoming HTTP connections to multiple servers, making sure that packets of the same session go to the same server. It is an application typical of datacenters. The redirections are done by changing the destination IP address of the requests flowing through the box. The requests traversing the proxy are NATed, to ensure that the packets go back through the box so the source IP address can be set back to the original destination address.

We compared a load-balancer implemeted using MiddleClick to HAProxy[84] in TCP mode and the NGINX[85] reverse proxy. The setup is similar to the NAT experiment. The proxy solution load-balances in a round-robin way the connections towards different IP addresses that are actually served by the same NGINX server. We made requests for file sizes from 0 KB to 256MB.The throughput can be seen in the first graph of figure 5.18. The "direct" line shows the performance of the testbed itself by removing completely the *DUT* and having the *generator* directly making requests to the *server*. To compare the performance of the various solutions in term of latency, figure 5.19 shows the average time to download one object according to the request rate.

Our solution outperforms HAProxy in term of throughput for every object sizes, ranging from a 3X improvement for small file sizes to a 2.5X improvement with bigger file sizes. In term of latency MiddleClick achieves an average file download time relatively close to the limit of the testbed.

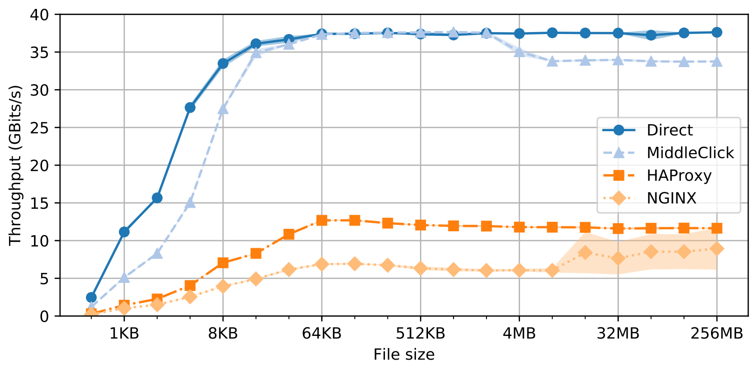


Figure 5.18: Data downloaded through a load-balancer using 128 concurrent connections executing as much request as possible to saturate the link. The proxy solution run on one core to constraint the performance by the CPU and not the test-bench capacity.

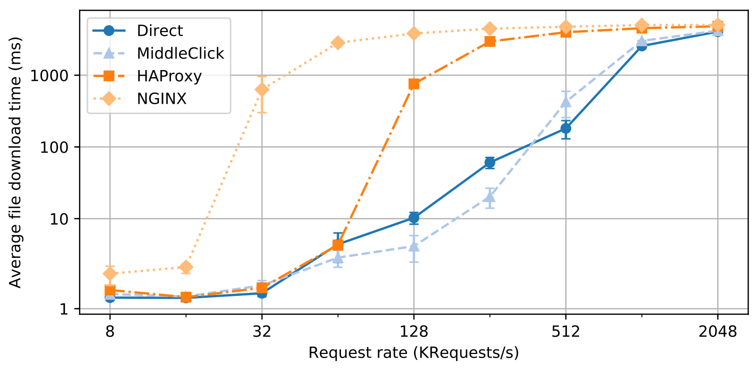


Figure 5.19: Data downloaded through a load-balancer using 128 concurrent connections to download a 1K file at an increasing request rate. The proxy solution run on one core.

5.7.4 Service chaining

Figure 5.20 shows the performance of running some service chains on 1 to 4 cores for 8K HTTP requests. First, we compare NAT implementations using MiddleClick and FastClick. Both of them actually achieve the limit of the testbed. That is 32 Gbps, using a single core of the *dut*. However, when adding a statistics VNF simply counting all bytes per-sessions, FastClick performances drop considerably because of the second session classification. MiddleClick, however, still achieves the limit of the testbed, as adding this function only extends the flow table per a few bytes. To further highlight the advantage of using MiddleClick, we introduce a few more functionalities to the chain. When adding TCP reconstruction, a hit is introduced that lowers performances to 20

Gbps, a cost paid for TCP state management and reordering of TCP packets. Adding VNFs for flow statistics (byte count per-session but only for the useful payload), a load balancer, and a computation of a checksum[1] induce very little impact as they will all have their space in the FCB at the same cost, extended to fit all VNFs of the service chain by MiddleClick without any manual tuning. Adding an IPS (simple string matcher) induce a bigger hit because of the pattern matching algorithm that must sill be improved. When using 2 cores, the chains up to the load balancer achieves 32 Gbps, while 3 cores are enough to run the chain up to the checksum.

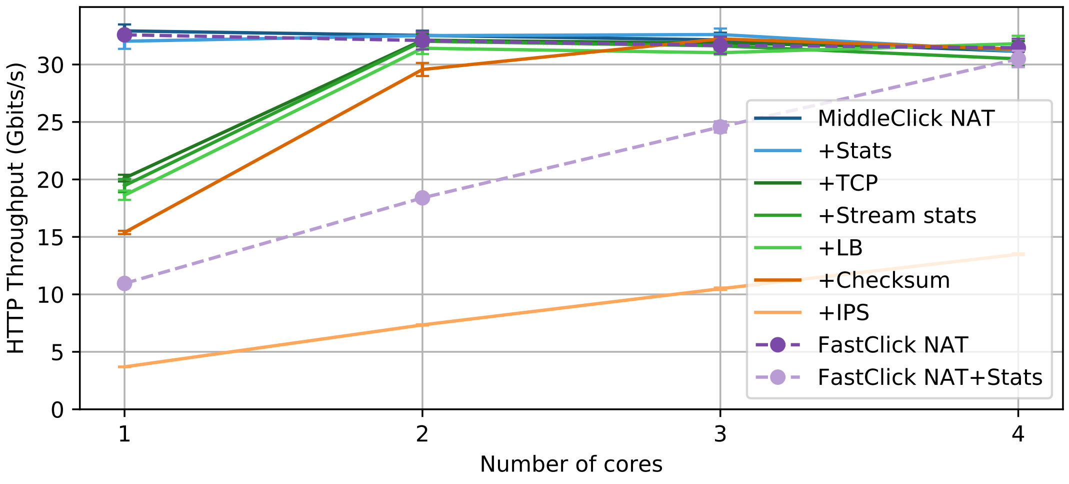


Figure 5.20: Impact of service chain length

Open Source Availability

MiddleClick is available at [48].

Try it out ! ■

[1]The checksum is a simple 4-byte-by-4-byte sum of the payload that could be used for signature matching. The checksum from previous packets is directly available in the FCB and the chunk iterator allows easy iteration over the payload of the stream to update the sum. When two chunks of payload are not on a 4-byte boundary, the last 1 to 3 bytes are saved in the FCB and added to the sum when the next packet arrives.

6

Cooperative infrastructure

In this chapter, we discuss how our middlebox platform can be integrated into a large scale network composed of *Software Defined Networking (SDN)*-enabled switches to do some pre-processing before the traffic hits the NFV servers.

Section 6.1 focuses on techniques to accelerate MiddleClick unified classification. We show how to take advantage of the consolidation of all classifiers inside MiddleClick to offload part or all the classification to available hardware such as modern NICs or a SDN infrastructure.

Section 6.2 tackles the problem of combining multiple VNF chains in a single box. Current approaches usually rely on software classification to dispatch packets among multiple processes or multiple virtual machines. Usually those systems use one or more cores to dispatch packets to other cores that actually execute the VNFs themselves[43, 107]. Other solutions use the NIC itself to send packets back and forth between VNFs, eventually leading to a bottleneck on the PCI Express link[109, 110]. We propose to use SDN switches to accelerate the classification between multiple VNF chains running on the same box while tagging the packets according to their traffic class. The offloading allows to reduce the work done by the server itself but most of all, the tagging avoids using CPU cores for the sole purpose of dispatching packets between isolated service chains. Indeed the NIC can strip the tag and direct packets towards one service queue per VNF, that will directly be opened and read by the isolated tenant.

This chapter in a nutshell

▶ **Context of this chapter**

- Recent NICs start to show some classification capabilities
- Datacenters have growing intelligence in the network infrastructure thanks to SDN-enabled switches
- Recent approaches use CPU to dispatch the packets to the right NFV application in software

▶ **Highlight of our main contributions in this chapter**

- Using the unified classifier of MiddleClick, the static classification part can be offloaded to upfront hardware
- The classification of multiple service chains can be aggregated by a controller to tag packets and allow them to be directly received by each isolated service chains

■

6.1 Traffic class classification offloading

It is possible to take advantage of MiddleClick's unified classification to offload it to some classification available on today's Smart NICs[103, 104], or SDN-enabled switches using *e.g.* the OpenFlow protocol.

The idea is to tag packets with *flow identifiers* when they enter the system. Those flow IDs are then used to dispatch the packets to the right middlebox components, and serve as keys to retrieve flow-related metadata (the corresponding FCBs in MiddleClick).

For example, an OpenFlow switch can be set up to replace the VLAN ID of packets according to the type of flow. An example match-action table for our running example from figure 5.6 is given in table 6.1. It is technically equivalent to table 5.1, where each FCB is simply numbered in order. A simple tag-to-FCB table can be used to avoid classifying in software.

This approach has some limitations, however:

- The VLAN ID size is only 12 bits, meaning only 4096 flow classes are available. If more classes are needed, a SDN switch capable of modifying larger packet fields (*e.g.* MPLS tag) or adding new bytes to packets must be used.

Match	Action
ETH_TYPE=IP, IP_PROTO=UDP	VLAN=0
ETH_TYPE=IP, IP_PROTO=TCP, TCP_DST=80	VLAN=1
ETH_TYPE=ARP, ARP_OP=0	VLAN=2
ETH_TYPE=ARP, ARP_OP=1	VLAN=3
*	Drop

Table 6.1: OpenFlow rules for tagging packets in the example of figure 5.6.

- It makes it harder to use VLANs in the network. However, a match on the initial VLAN ID can be done, provided that not too many VLANs are used. *E.g.* if the HTTP middlebox in our example was also discriminating traffic based on the VLAN, and there were four VLANs, we could simply duplicate the HTTP rule four times, once per original VLAN (*i.e.* rather than having one unique flow ID for all HTTP packets, we would have one per original VLAN ID).

- We are limited by the matching capacities of the upfront SDN switch. If we need to classify on unsupported packet fields, we can still classify as much as we can on the SDN switch, and only do the remaining classification tasks in software.

- Finally, and more importantly, current OpenFlow switches can only deal with static classification, not session handling. Having a separate flow ID per session (*e.g.* TCP 4-tuple) would require a controller to add new rules dynamically whenever new sessions are seen, that would be equivalent to migrate the MiddleClick classification to the controller. Moreover, currently there are no OpenFlow switches that can handle millions of OpenFlow flow creation or modification messages. Hopefully, SDN switches might gain the ability to directly create new rules on the fly in the future (*e.g.* see the DevoFlow proposal [111]).

One can even avoid classifying in software the first packet of a flow using SmartNICs capabilities to run the classification inside the NIC itself, and dispatch packets to queue according to their traffic classes. A packet from a queue must match one of the corresponding rules, reducing the classification work, or even avoid it entirely if only one rule leads to that queue. The limiting factor here is the number of available queues, and the matching capacities of the NICs. What we would want to see in future hardware, is the possibility to ensure that a queue will correspond to a single flow. But while our Intel X520 NICs have FlowDirector, a mechanism to map classification rules to different NIC queues, the masks for the fields are global, meaning that if one flow specification has a wildcard (*e.g.* on a source port), each other flow specification will use the same wildcard. Having the classification unified at the entry of the box could also allow us to use Intel's Flow Director as a cache like in [112] with a gain of 40% in classification throughput using an heuristic to cache elephant flows. However regarding FlowDirector, Intel is evolving in the opposite direction, as recent NICs do not support masking on standard IP, TCP our UDP fields anymore, leading to very limited unique-field classification.

6.2 Service chain classification offloading

Contribution notice

Most of section 6.2 is directly relating part of “Metron: NFV Service Chains at the True Speed of the Underlying Hardware”[25], our publication presented at NSDI'18, resulting from a collaboration between Georgios Katsikas (RISE SICS), Tom Barbette (University of Liege), Dejan Kostić (KTH Royal Institute of Technology), Rebecca Steinert (RISE SICS) and Gerald Q. Maguire Jr. (KTH Royal Institute of Technology).

One man cannot solve all the world's problems ■

For efficiency and to allow consolidation, it is best to allow combining multiple service chains inside the same NFV dataplane agent. A unique server can then run multiple service chains and scale or combine them to reduce energy consumption.

The solution we propose is to use a SDN switch to implement the service chain selection, that could possibly run on different NFV servers. The switch directs the packets towards the right server, but also tags packets so the NFV server can directly dispatch packets to the right service chain *in hardware*. Each NFV server runs a master agent that is responsible of advertising the tagging methods that its NIC support (*e.g.* VLAN, MPLS, ...) to a controller that can then program the SDN switch accordingly. The master agent is responsible for bringing up the service chains when the controller sends a request to launch a new service chain. The master agent configures the NIC according to the tags that the controller chose as stated in the request, which will be from one of the tagging methods from the list of supported tagging methods the master agent previously advertised.

This chain of action is depicted in figure 6.1.

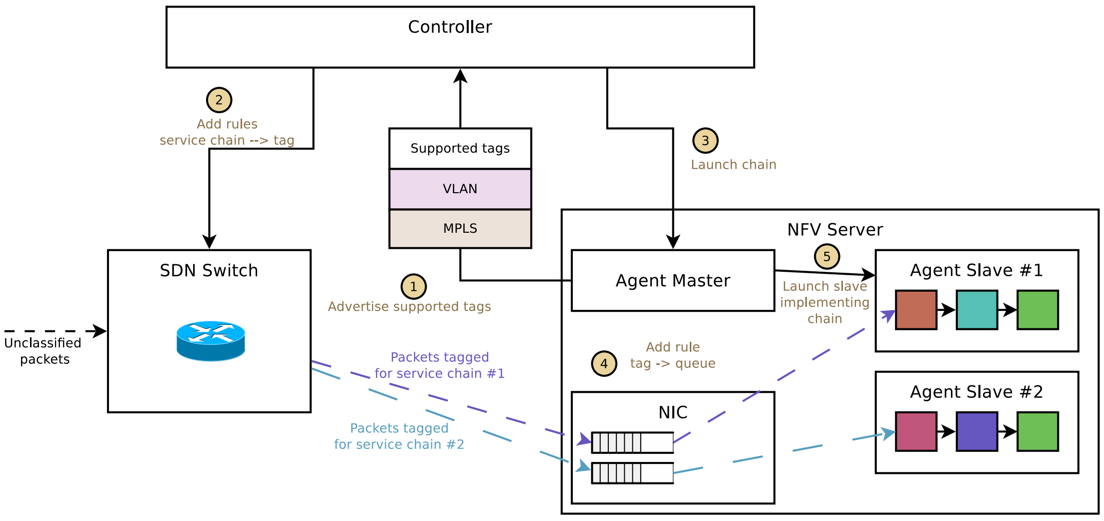


Figure 6.1: Architecture to use tagging to dispatch packets to multiple service chains running inside a single NFV server

When a dataplane agent is launched, the agent advertises the available NICs and CPU cores of the NFV server it controls. The agent also advertises the tagging capabilities of the NIC (figure 6.1 - 1), *e.g.* the ability to directly send packets with a specific VLAN to a hardware queue. For example, Intel 82599 chipsets support tagging using ethernet MAC addresses and/or VLAN.

The controller can then choose one of the supported tagging methods to insert rules inside the upfront SDN switch to tag packets for a specific service chain and to directs them towards the right NFV server (figure 6.1 - 2).

The controller can then ask the master agent to launch the service chain software. The service chain may run directly in the same process, in a new child process or in a fully virtual machines. We'll refer to a tenant running one of the service chains as a slave agent. Most NICs are now able to expose a subset of their hardware queues to a children process (*e.g.* using VMDq with our Intel NICs) or virtual machines (using PCIe *Virtual Functions (VFs)*). In contrast, the slave could also run inside the same process context if isolation is provided by safe-language checkings like NetBricks[42].

The master agent can then instruct the NIC to direct packets with the service chain's tags to a specific hardware queue (figure 6.1 - 4) so the slave agent can directly access the queue without any packet copy or VM entry/exit if using virtualization.

The master can finally launch the slave that implements the corresponding service chain itself (figure 6.1 - 5). In the example from figure 6.1, we show the master/slave process approach. The slave agent is only accessing its subset of queues knowing that, thanks to the tagging made by the SDN switch, all packets in its dedicated queues are of the traffic class it has to handle. This technique could possibly be combined with the idea in section 6.1, sending multiple tags to the service chain that each refer to specific traffic classes destined to the very same service chain but for different components. For example, one tag/queue for HTTP traffic, one tag/queue for UDP traffic, ...

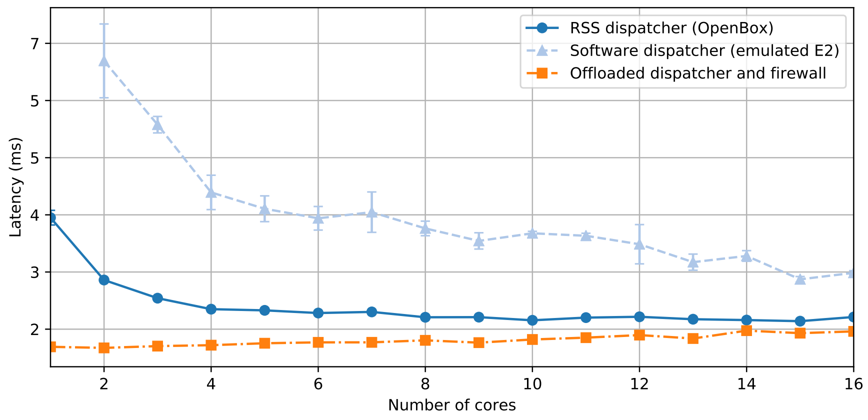
The key differentiator between this approach and earlier NFV work is the tagging module shown in figure 6.1. This module exposes a map with tag types and values that each NIC can use to interact with each CPU core of a server and this information is advertised to the controller. The controller can remotely and *dynamically* associate traffic classes to specific tags in order to enforce a specific traffic class affinity, thus controlling the distribution of the load. More importantly, this traffic steering mechanism is applied by the hardware (*i.e.* NICs), hence tadditional CPU cores are not used for classification in the dataplane agent (as E2 [43] does) to perform this task, thus packets are directly dispatched to the CPU core that executes their specific packet processing graph.

6.2.1 Evaluation

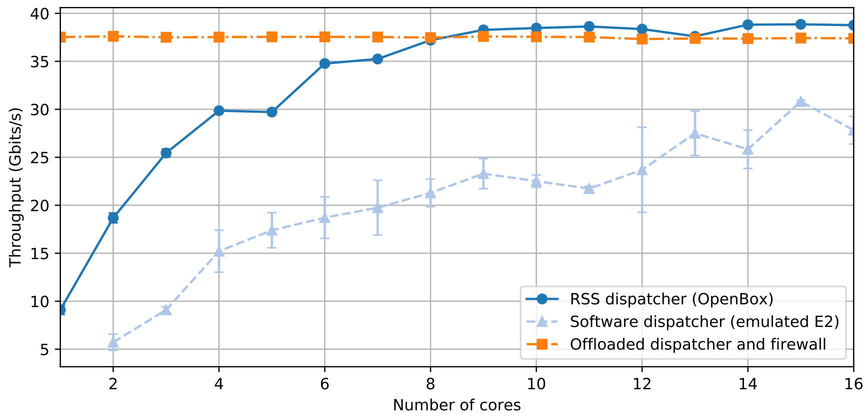
To validate the idea and measure how much performance could be gained from avoiding software service chain classification inside the NFV server itself, we made a static experiment without the controller, manually assigning tags and launching rules.

For the data plane, we implemented the agent on top of MiddleClick and we used the Virtual Machine Device Queues (VMDq) filters of DPDK 17.02 to implement the hardware dispatching based on the values of the destination MAC address and VLAN ID fields. Our prototype uses the destination MAC address as a filter, because the large address space of this header field provides unique tags for trillions of service chains.

Our testbed consists of 2 identical machines, each with a dual socket 16-core Intel® Xeon® CPU E5-2667 v3 clocked at 3.20 GHz. The cache sizes are: 2x32 KB L1 (instruction and data caches), 256 KB L2, and 20 MB L3. Hyper-threading is disabled and the OS is the Ubuntu 16.04.2 distribution with Linux kernel v.4.4. Each machine has two dual-port 10 GbE Intel 82599 ES NICs. We use a NoviFlow 1132 OpenFlow switch



(a) Latency



(b) Throughput

Figure 6.2: Performance of a campus firewall with 1000 rules at 40 Gbps, using our system, (*ii*) an accelerated version of OpenBox using RSS, (*iii*) a software-based dispatcher that emulates the performance of E2.

with firmware version NW400.2.2 and we attach the two machines to this switch. The 4 ports of the first machine are connected to the first 4 ports of the switch to inject traffic at 40 Gbps.

To test the overall system performance at scale, we deploy a service chain of a campus firewall, followed by a DPI. The firewall implements access control using a list of 1000 rules, generated using the traces to ensure all traffic is accepted. The second VNF deeply inspects the output of the firewall using a set of regular expressions similar to Snort (taken by [41]).

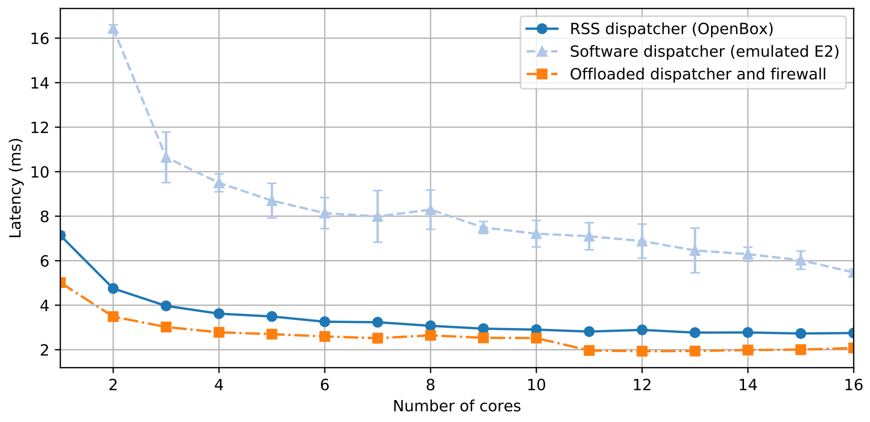
We compare our approach against two state of the art systems. Specifically, an accelerated version of OpenBox based on RSS and an emulated version of E2 [43]. In the latter case we emulate E2's SoftNIC by using a dedicated CPU core that dispatches packets to the remaining CPU cores of the system (1-15), where the service chains are executed. [1]

We injected a real campus trace that exercises all the rules of the firewall at 40 Gbps and measured the performance of the three approaches. Figure 6.2 visualizes the results.

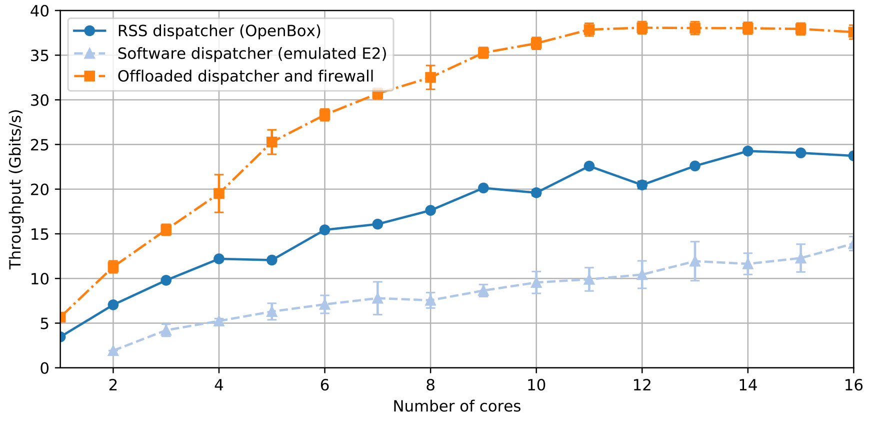
First, we deploy only the firewall VNF of this service chain to quantify the overhead of running this VNF in software, as compared to an offloaded firewall. Indeed, the controller will place all the classification inside the SDN switch, leaving no processing to do for the VNF. To fairly compare our approach against the other two approaches, we start a simple forwarding VNF in the server, such that all packets follow the exact same path (generator, switch, server, switch, and generator) in all of the three experiments.

Figure 6.2b shows that OpenBox and the emulated E2 can achieve this large firewall at line-rate. However, this is only possible if half (or more) of the server's CPU cores are utilized. Specifically, OpenBox requires 8 cores, while the emulated E2 requires 2 additional cores. In contrast, our approach can totally offload the firewall rules to the switch, which can easily realize the access control list (ACL) at line-rate, thus one CPU core at the server is enough to achieve the maximum throughput.

[1]The graphs of the emulated E2 in Figure 6.2 start from core 2 because the first core is reserved for dispatching traffic to the VNFs.



(a) Latency



(b) Throughput

Figure 6.3: Performance of a campus firewall with 1000 rules followed by a DPI at 40 Gbps, using: (*i*) Our system, (*ii*) an accelerated version of OpenBox using RSS, (*iii*) a software-based dispatcher that emulates the performance of E2.

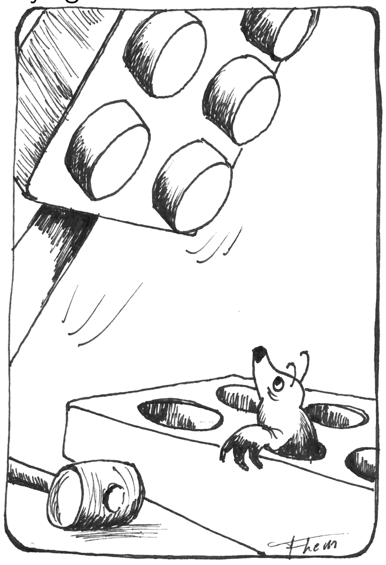
Looking at the latency of the three approaches (Figure 6.2a), it becomes evident that software-based dispatching (yellow solid triangles) inflicts a large amount of unnecessary latency. Hardware dispatching using RSS (green solid circles) achieves substantially lower latency because it avoids inter-core communication. However, since the firewall introduces heavy classification in software, OpenBox still exhibits high latency that cannot be decreased by simply increasing the number of cores. Specifically, using 16 CPU cores achieves comparable latency with just 4 cores. In contrast, our approach achieves constant low latency by exploiting the switch's capability to match large number of rules at line-rate. This latency is 2.7-4.3x lower than the latency achieved by the emulated E2 and OpenBox respectively, when each system uses only one CPU core for processing the VNF (emulated E2 requires 2 in this case). At the full capacity of the server, the latency among the three systems is comparable; our approach outperforms the emulated E2 and OpenBox by 6% and 19%.

Next, we chain this firewall with a DPI VNF in order to realize the entire service chain. This chaining further pushes the performance limits of the three approaches as shown by the dashed lines in Figure 6.3. In this case, our system implements the DPI in software.

Our approach exploits the joint network and server capacity to scale even complex VNFs, such as DPI, at line-rate (red dashed squares in Figure 6.3b). Most importantly, our approach requires only 11 CPU cores on a single machine to achieve 40G, thus *substantially shifting the scaling point for larger service chains.* The latency results further highlight the system abilities. With 11 CPU cores, the server deeply inspects the packets for this service chain by inflicting only 17% more latency than the latency required to realize only the firewall. At the same time, OpenBox and the emulated E2 inflict 40-96% more latency than our approach, with almost half of the throughput. This difference is growing more rapidly when fewer CPU cores are utilized. For example, when each system uses one CPU core our approach achieves 67-327% lower latency than OpenBox and the emulated E2 respectively.

As we explained in our approach (see Section 6.2), the secret in the system ability to scale complex VNFs (*e.g.* DPI) lies in the way that the incoming traffic classes are tagged and then dispatched to the right CPU cores.

Trying all combinations at once



7

Experimental automation and reproducibility

In this chapter we present a Network Performance Framework (NPF) which allows replaying tests automatically, from software download and compilation up to ready-for-paper graphs.

NPF is based on comprehensive test description files, which describe how to run a single test. They describe which software to run, how and where to run them. Each test comes with sets of parameters to try (*e.g.* number of threads, buffers size, packet length, packet rate, . . .), that NPF can use to compare performance metrics (*e.g.* throughput, delay, . . .) of different programs (or versions of a program), for all possible combinations of parameters.

NPF takes care of launching the involved software on a *Device Under Test (DUT)* and some packet generator on another computer. NPF launches software across a cluster, replacing MAC, IP addresses or PCIe addresses in the configuration lines according to the NICs involved. It is able to change NICs drivers to use DPDK, Netmap or the normal drivers.

By sharing their test files, researchers not only provide a way to rebuild automatically all the graphs in their papers, but they also share the much larger range of settings they tried to convince themselves their results were fair, using built-in statistical and machine learning tools.

Most of the graphs presented in this thesis are actually available as NPF experiments. Therefore most experiments can be reproduced by invoking a single NPF line, describing the testbed and which experiment to launch. One only needs two or three computers according to the experiment, interconnected by high-speed NICs.

This chapter in a nutshell

▶ **Context of this chapter**

- There are few experiment manager to organize testing in the context of high-speed networking. While most networking topics can be simulated or virtualized, high-speed networking requires real hardware access.
- The literature in the high speed networking topic presents a generalized reproducibility problem[113, 114, 115].

▶ **Highlight of our main contributions in this chapter**

- We build NPF, a tool to automate a test case with many parameters to vary.
- Tests can run across a cluster automatically, changing addresses and parameters according to the environment. That eases the reproducibility problem.
- NPF allows to quickly get insight into understanding the many results of an experiment using an automatic graphing system and statistical tools.

■

7.1 Network Performance Framework

Frameworks for network test definition and execution are usually based on emulation or virtualization. Both of those are not suitable for high-speed networking tests as they induce either severe performance hits or involve hardware-specific features.

This leads to a plethora of recent papers where the description of the tests behind numbers and figures are in the better cases barely comprehensive scripts to be executed on some not well-defined cluster of nodes. It is not rare that the code is not even available, or missing pieces.

[113] argues in favour of moving from repeatable research to replayable research, which is still an open problem that NPF tries to tackle.

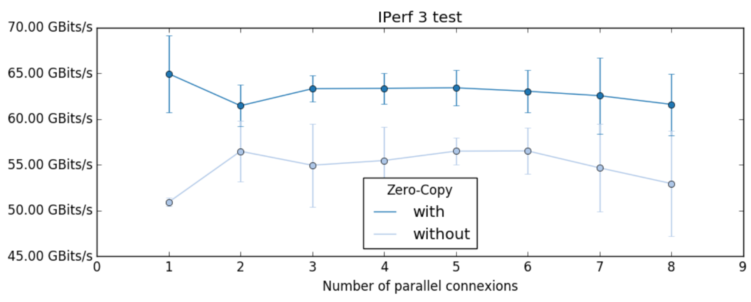
NPF is an orchestrator to be used to reproduce multiple self-contained tests across a set of nodes. NPF allows to easily run the same tests on different hardware and configurations and as a result ensures robustness of the tested solution. However, it does not support describing or ensuring the physical layout behind the tests.

```
%info
IPerf 3 test
This tests measures the throughput of a local TCP connection
%variables
PARALLEL=[1-8]
ZEROCOPY={:without,-Z:with}
%config
var_names={PARALLEL:Number of parallel connexions,ZEROCOPY:Zero-Copy}
%script
iperf3 -s &> /dev/null
%script
echo "RESULT $(iperf3 -c localhost -P $PARALLEL $ZEROCOPY \
    | tail -n 3 | grep -ioE "[0-9.]+ kbits")"
```

(a) The test description file.

./npf-run.py iperf –testie tests/tcp/01-iperf.testie.

(b) The command to launch NPF on this testie file.



(c) Automatically generated graph displaying the results of the experiment after being launched with (b)

Figure 7.1: Simple configuration to test TCP throughput of a local connection using iPerf3, and the resulting graph produced by NPF.

NPF can download, compile and distribute software under test across a cluster, or on a single node. It will handle a series of scripts to run across nodes, and re-execute the test under various different settings as described by a single *testie* file. Testies are freely inspired by the Click Modular Router[11] test suite, but NPF is suited to test performance of any networking software such as middleboxes or NFV platforms.

Figure 7.1 (a) shows a first simple local testie that will run iPerf3 to evaluate the performance we can expect from a localhost TCP connection, using different parallelism parameters, both with and without the zero-copy feature of Linux TCP sockets. The resulting graph as produced by NPF can be seen in figure 7.1 (c), automatically generated after launching the experiment, *i.e.* with the command in figure 7.1 (b). Definition of testies is further discussed in section 7.2.

For strict result reproducibility, we expect NPF to be used in conjunction with physical testbeds platforms such as Emulab[116] or PlanetLab[117]. Those give access to strictly defined nodes and their topology (either local or global), but do not provide the ability to describe and manage the tests themselves up to the graph generation and analysis we present in this chapter, nor do they provide a way to share the tests configurations and parameters in an easily deployable and shareable fashion.

In the same spirit, virtual environments like Mininet[118], Mininet-HiFi[119] or Netkit[120] allow to virtualize the testbed. Some others emulate it, as the the NS simulator[121] does. Both solutions are fine for functional verifications and to get a first glimpse into performance, but introduce slowdowns which prevent ground truth when trying to study the behaviour of high-speed software. Virtualization and emulation often prevent running tests that rely on hardware, and studies specific features and their limitations (*e.g.* how the NIC behave at multi-10Gbps, hardware offloading capabilities, multi-queues load-balancing, . . .). Those also limit the testing of software requiring kernel bypass, like the DPDK[31] and Netmap[32] frameworks that are often used as an essential brick for high-speed software networking.

NEPI[122] allowed to abstract the experiment environment, using a python interface. We believe our framework is easier to use, as the most basics test description file is only a list of bash lines that researchers would usually manually launch on different computers. One can then add support for multiple variables, automatic binding, roles for cluster definition, etc. . . NPF has virtually no learning curve. We believe it is therefore much easier to adopt than other systems because it allows for a progressive adoption. The syntax of variables makes it very easy to automate the experiment for different values and produce graphs automatically, a feature not provided by NEPI.

Section 7.3 discusses the multiple NPF tools allowing to understand the data generated when many variables are involved, *i.e.* mainly the relation between the parameters and the performance.

7.2 Architecture

NPF itself is written in Python 3 and relies on one *testie* configuration file to describe a single test. NPF comes with a set of *repo* files describing how to fetch and build multiple networking software. We expect the set of *repo* files to grow so that researchers only need to share their testies files.

7.2.1 Testie

Testies are organised in sections, of which a complete definition can be found at [52]. In a nutshell, the `%variables` section defines a list of variables and their possible values, given as a list, an interval, a logarithmic interval, ... up to the ability to generate a list of firewall rules. NPF will replace all `$variable` occurrences in the `%script` sections, which define bash commands to run for the test. Variables will also be replaced in a `%file` sections describing some files to create before running the scripts. While the script language being bash may seems a bit limiting, a simple line is usually sufficient to compile or build another program with its source defined in a `%file` section. Table 7.1 lists the available sections.

7.2.1.1 Variables expansion

NPF will parse the testie file, and call the handler of each section implemented as *Section* objects. The **%variables** sections understand multiple kind of variables, like linear increasing ranges, exponential ranges, list, ... Each variable type is associated to a *Variable* object that defines a regex to be tried on each line of the **%variables** section. The *Variable* objects implement a *makeValues()* function that will return the list of possible values for the variable. The execution function will iterate through each variables, re-creating the the **%file** and re-executing **%script** sections for each possible combination of the values.

7.2.1.2 Initialization and pre-defined scripts

%init sections allow to execute some special scripts before any of the runs start. This allows to initialize NICs, set IP addresses, routing tables, system options, Linux TCP settings, ... As most of those **%init** sections are actually identical across tests, NPF introduce the **%import** section that allows to import other *module* testie that execute specific tasks. A module is like any other testie, with *init* and *script* sections.

Section	Description	Example
info	Title of the test on the first line, followed by the human-readable description of what it does.	**%info** A simple test This is a simple example testie that will [...]

config	Define configuration options, like the number of runs to execute, informations about the units, the color of the lines in the graph, ... All configurations options are optional.	**%config**
		n_retry=1
		n_runs=3
		var_name={BURST:Number of packets}
		var_format={BURST:%d}

variables	Defines variables that will be replaced in the script and file sections. The test will be executed for all set of values resulting from the crossproduct of all variables	**%variables**
		PARALLEL=[1-8] //1,2,3, ... 8
		FILE_SIZE=[1*1024] //1,2,4, ... 1024
		ODD=[1-17#2] //1,3,5, ... 17

file NAME	Create a file named NAME, with the content of the section. Variables in the file will be replaced by their values	**%file** ODD Here is an odd value : $ODD
script	Executes a bash script, its content is also searched for occurence of variables. Scripts can have some specific parameters, like autokill=true which will stop all scripts when the given one finishes, or delay to execute it after some time.	**%script** autokill=true delay=2 echo "Downloading a file of $FILE_SIZE KB..." wrk http://$HOST/$FILE_SIZE -c $PARALLEL
init	Similar to script, but executed before running a test.	**%init** ifconfig ${server:0:ifname} promisc

late_variables	Allows to define variables that will be created at each run of the test. This allows to generate part of files or scripts according to the value of the variable for the current test. For example, duplicate some line of a configuration file according to the number of CPU used in the test.	**%late_variables**
		MULTITHREAD=$((1 if $CPU >1 else 0))
		PIPELINE=DUPLICATE(CPU , pCPU :: Pipeliner() ->myElement;)

import MODULE	Import a "module" testie. Modules are normal testie files but intended to be imported in another script. Modules generally implement traffic generators, allows to bind devices to drivers, ...	**%import** fastclick-replay-single trace=trace.pcap

Table 7.1: Testie file sections

click.testie

```
name=Click
branch=master
url=https://github.com/kohler/click.git
method=git
bin_folder=bin
bin_name=click
configure=./configure --disable-linuxmodule
  --enable-userlevel --enable-user-multithread
tags=click,vanilla
```

fastclick.testie

```
parent=click
name=FastClick
url=https://github.com/tbarbette/
  fastclick.git
configure+=--enable-batch --enable-dpdk
tags+=fastclick,batching,dpdk
```

Figure 7.2: Example of software source definition

7.2.1.3 Results parsing

NPF will parse the output of script execution searching for "`RESULT [0-9.]+`" and some standard units suffix (*e.g.* bps, Gbps, ms, us, μs, ...), or a user-defined pattern. It will also re-run each test multiple times, as defined by the `n_runs` configuration variable given in the **%config** section (defaults to 3), to compute and plot standard deviation and ensure test stability for the given set of variable values.

7.2.1.4 Regression

NPF includes a regression tool able to compare performance of multiple software versions, ensuring that one version did not break the performance. The **%config** section has also multiple parameters to tweak the variance the regression tool will accept, reject outlying values, or run the test again multiple times before rejecting a new version of the software and mailing the author about a possible regression.

7.2.2 Software

Testie files are executed against a specific software in either a given version, or its latest version if not specified. Currently, two download/install methods are supported: `git` and HTTP. The git method is able to go back in history to review performance changes across versions automatically. Using git, NPF can watch for new commits and send a regression report to some people and the commit author.

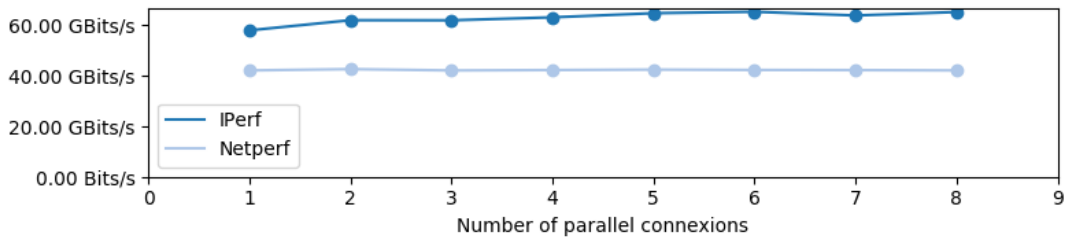
Figure 7.2 shows the *repo* definition for the Click Modular Router[11], and demonstrates how to extend a configuration file using the `parent` keyword with FastClick[54], which is based upon Click and can keep most of its parameters. Most of the examples here use Click and its variants as Click has been the basis of most our recent work. However NPF is not

```
%variables
PARALLEL=[1*8]
ZEROCOPY={:without,-Z:with}

%iperf3:script@server
iperf3 -s &> /dev/null
%iperf3:script@client delay=1
echo "RESULT $(iperf3 -c ${server:0:ip} -P $PARALLEL $ZEROCOPY \
    | tail -n 3 | grep -ioE "[0-9.]+ kbits")"

%netperf:script@server
netserver -D -4 &> /dev/null
%netperf:script@client delay=1
echo "RESULT $(netperf -f kbits -l 2 -n $PARALLEL -v 0 -P 0  ${server:0:ip}kbits"
```

(a) Main parts of a testie that compares multiple different pieces of software by using tags, and runs client and server scripts on remote nodes using role definitions.



(b) Output graph.

Figure 7.3: Advanced configuration supporting remote execution and multiple software

limited to Click, one can use the Moongen[123] generator to test the performance of any networking software, for instance.

Repo files may also define a list of tags, allowing to use some specific configuration, script, files or variables in the testie file according to the software under test. Figure 7.3 (a) shows how tags are used to invoke the right script when comparing iPerf and Netperf. The NPF comparator tool uses this feature to compare multiple piece of software against each other as depicted in figure 7.3 (b).

7.2.3 Cluster

All `%script` sections of testies can define a role they should be associated to, intended to run on specific nodes. Example roles are client (or packet generator), dut (device under test), and server (or traffic sink).

Roles such as client and server in figure 7.3 are mapped to real nodes using the NPF `--cluster` argument followed by mapping parameters like

```
npf-run.py --testie my.testie --cluster client=node-01
    server=tom@node-02.ulg.ac.be:/home/tom/npf
```

In this example, the client role maps to a specific `node` configuration file that will define *node-01*. The server role maps directly to a node. The `node` file defines some variables such as the IP and MAC addresses of each dataplane NIC of the given node. `%script` or `%file` sections can use special variables that will be replaced by other nodes IP or MAC addresses to allow dynamic connection between roles without knowing the nodes specifics a priori. In figure 7.3 (a) the ${server:0:ip} variable will be replaced by the IP of the first NIC of the node taking the server role.

Communication between NPF and the nodes is done through SSH. Undefined roles run locally, so that in the configuration of figure 7.3, the iPerf test is identical to the one in figure 7.1 when roles are not mapped through the `--cluster` argument.

7.2.4 Multiple metrics

NPF is able to understand multiple results from an experiment. The proposed technique is to output lines such as "RESULT-TYPE VALUE unit" where TYPE is the result type (latency, throughput, number of packets per seconds, ...). Alternatively multiple regular expressions can be defined in the testie file.

7.2.5 Data representation

Figure 7.4 show the representation of results in memory. The dataset is kept in memory as a dictionary data structure of *Run* keys mapping results. A *Run* object is a list of variables as defined in the **%variables** sections, and the values fixed for a specific run. Results are themselves kept as a dictionary of TYPEs as keys with the list of values returned by the experiment for the given type. Individuals values are not aggregated (*e.g.* as an average of values per run) to allow the user to ask for specifics statistical metrics such as the median, compute the standard deviation, ...

NPF includes a result cache. After each execution, the dataset will be stored in a file, per test, per software under test and per version. When re-executing a test, the cache is

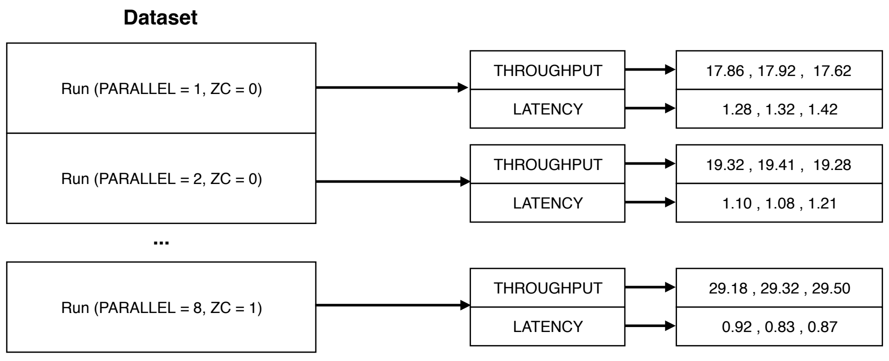


Figure 7.4: Internal representation of the results of a full execution

looked for existing results for the same testie file, with the exact same software, version, and individual *Run* parameters. This allows to avoid re-executing the whole test when adding a few values to the parameters. For instance, after trying an experiment using a single CPU core, re-executing the experiment for 1 to 4 cores will only launch the tests for 2 to 4 cores.

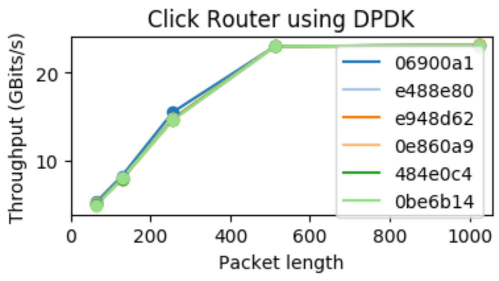
7.3 Interpretation of results

7.3.1 Graphing

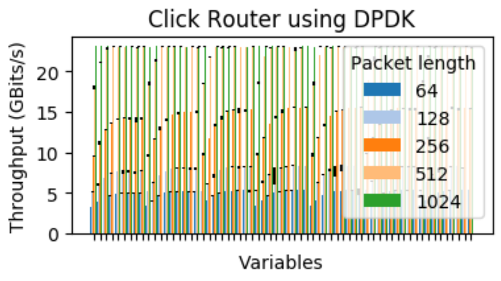
A graph is always automatically generated for each test, using a line plot or a bar plot, grouping variables as different series of the graph when needed to make the results understandable as quickly as possible. We use matplotlib for graph creation. Figure 7.5a, 7.5b and 7.5d show result of the grapher under different configuration for the same Click router test case.

7.3.1.1 Data transformations

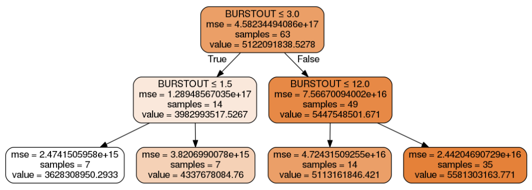
When comparing multiple versions of a program (regression tests), or multiple different software solutions described by a single testie (software comparison), NPF re-executes the whole testie file multiple times, meaning that one dataset is returned per executions. We'll use datasets as series of the graph to produce. We then count the amount of *dynamic* variables that is variables that have multiple values in the datasets, as the user may have fixed some of the variables to a given value. If there is only one database, one of the dynamic variables will be extracted from the dataset and we'll duplicate the database per value of the variable, removing the variable from the *Run*s along the way. If there is no dynamic variable left, a barplot will be produced, showing the results for each of the series. If there is



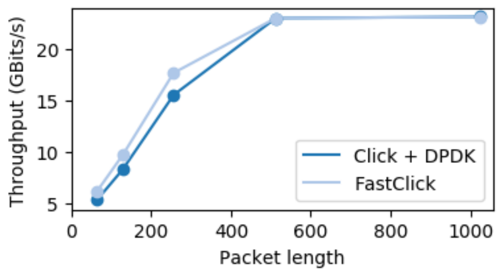
(a) Regression test for Click



(b) Result for the full 315 possible parameters combinations



(c) Regression tree visualisation for packet of 64 bytes



(d) Click compared to FastClick

Figure 7.5: Multiple figures as produced by NPF tools for the same Click-based router testie using DPDK for I/O

only one, a lineplot will be used as in figure 7.5a. If there are more, a barplot will be used, grouping variables as different bars such as in figure 7.5b.

One graph is produced per-type of result, *e.g.* one for throughput, one for latency, ... The user may combine results in a single graph using dual axes, or transform the results as a dynamic variables (*e.g.* RESULT=throughput, RESULT=latency, . . .).

This allows the user to get automatically a glimpse into the results, with many other options to tweak the graph.

7.3.2 Output module

The datasets produced by the test can be exported as CSV files. The default parameter is to print the average and standard deviation of the results in columns, with each runs as rows. The user can ask for many metrics using the "–output-columns" argument such as median, percantiles, all results using one column per results, . . .

7.3.3 Statistical analysis

In a real-life scenario, the number of parameters can quickly grow leading to an amount of data that cannot be depicted well enough in a graph. Figure 7.5b displays 315 different configurations of the router test case varying burst parameters and packet length. To help finding the most interesting parameters NPF will give worst and best cases, and the average when results are grouped per-variable.

NPF builds regression trees using *skikit-learn*[124] to compute the importance of each variable, and decide which variables can be fixed in subsequent runs because they do not influence the results.

The regression tree can also be directly visualized like in figure 7.5c. Each node can be interpreted as an important pivot for a given variable, separating performance results in branches that will lead to the most entropy.

The command line also allows to fix parameter values to re-execute the test and advance from deductions to deductions while keeping best or interesting values only. After executing a test with many values per variables, the results will be stored in the NPF cache as discussed in section 7.2.5. Therefore reducing the parameter matrix size is instantaneous as results will be in the cache.

For our own usage during development phase, we often let NPF run one night on a very big matrix of parameters with all parameters that could potentially impact performance, and quickly analyse results in the morning using the statistical insight. As any parameter combination is already in the cache, we can quickly try to fix some variables and see how the device under test performed from different angles.

7.4 Conclusion

Given a (perhaps virtual) cluster, NPF executes a test across its nodes, and automate it for each line of a matrix of parameter values to study a given software under different configurations. While existing testbed handlers (either physical, virtual or emulated) usually allow to run some scripts and restart them, they lack support for unifying test case definition, and lack tools to quickly get a glimpse of the impact of some parameters. NPF provides multiple tools such as a comprehensive grapher, and use machine learning techniques to help understand the classes of performance when many parameters are involved.

NPF is particularly suited for high-speed packet processing experiments. It supports automatic binding of NICs to DPDK or Netmap, and has multiple generic scripts, called *modules*, to generate worloads using multiple generation software such as WRK[108], Apache BenchTools[125], Iperf[126], Netperf[127] and many different FastClick configurations. We provide FastClick configuration to generate different kind of workloads, and replay traces pre-loaded in memory at more than 150Gbits/s while measuring latency by tagging packets of the trace.

NPF comes with support for downloading, building and installing software and distribute them over a cluster. Testies can integrate a configuration phase to set up network requirements such as IP addresses or OS configuration.

As a result, NPF can also improve the reproducibility of network publication results, by inviting researchers to share their testie files. Reviewers would be able to run the same tests quickly; first locally, then on a similar but slightly different cluster, and maybe on a public testbed to ensure a very good reproducibility.

NPF is available at [52]. It is distributed under the GPL3 license and welcomes merging your own tests to build a big testie and repo files database, truly enabling sharing tests in as few lines of code as possible.

Open Source Availability

NPF is available at [52].

Try it out ! ■

8

Conclusion and future work

The task of proposing a programmable platform for network infrastructure has been performed in a bottom-up approach.

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[32] and the DPDK[31]. We showed those frameworks were needed for fast I/O because of the current inadequacy of Kernel networking stack for *pass-through* workloads. We proposed proof-of-concepts improvement to the Linux kernel to demonstrate that the relative slowness of NFV applications in userlevel is not a state of affairs. The Kernel may evolve, allowing to get rid of frameworks like DPDK or Netmap. On top of high-speed I/O proposals, we studied some other kernel system calls that slowed down performance in real appliances and also benefit from Kernel by-pass, such as getting the time in userland. Future work may also review userlevel safe random number generation and more efficient IPC, which still imposes a performance penalty when used.

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. We reviewed distributed processing models and efficient data structures for parallel processing. We reviewed when to use different solutions for both topics, and provided helpful implementations in FastClick.

On top of FastClick, we have developed a high-speed framework to build service chains of middleboxes. Our system has better throughput and latency than other compared approaches, thanks to the avoidance of multiple reclassification of packets as they pass through the various middleboxes in a chain. It also easily enables the offloading of part or all of the classification to dedicated hardware, further improving performance.

Our framework eases the handling of per-traffic class and per-session state. The middlebox developer can specify, in a flexible way, which traffic class the middlebox is interested in, the sessions and the size of the state it needs for each session. Then, the system automatically delivers packets for the given traffic classes, and provides and manages the associated session storage, which is directly available to the middlebox through the packet themselves (thanks to the metadata carried along with the packets).

Finally, our framework exposes simple stream abstractions, providing easy inspection and modification of flow content at any protocol level. The developer only needs to focus on the middlebox functionality at the desired protocol level, and the framework will adjust the lower-level protocol headers as needed. Our framework can act as a man-in-the-middle for TCP connections, greatly simplifying high-level middleboxes development, while avoiding the overhead of a full TCP stack.

The architecture we proposed for flow stalling and on-the-fly modifications should generalize to new or unknown middleboxes and protocols thanks to our use of a "stacking" approach. Protocol contexts can be chained one after the other, each layer taking care of the implications of given requests, before passing the requests to the lower layers. On top of contexts, session definitions are completely unrestricted and a new protocol using different ways to map a session from the 4 TCP tuples could be implemented very easily. This removes one hard part of designing a protocol implementation for middleboxes with *e.g.* hash tables or other similar dictionary structures.

Our open-source implementation, codenamed MiddleClick, shows significant performance improvements over traditional approaches on a few test cases. The deep understanding of low-level considerations and the focus on performances makes it one of the fastest NFV dataplane agent publicly available to date.

Currently, the classification consolidation of MiddleClick is done when it launches and is then kept as is. We would like to allow more dynamicity as future work, but that will also require core modification to Click which currently relies more on a full graph replacement than surgical element insertion and removal.

The unified classification is, of course, limited. While in MiddleClick VNF components can expose any rule, the classifier may be less suited to particular classification such as core routers forwarding table with thousands of routing prefixes that will benefit from special purpose algorithm. The general rule of thumb to know if a rule should be exposed to the system is to ask if that rule may also be spawned by another VNF, and therefore benefit from being *factorized* by the MiddleClick flow manager. Maybe we could go further than factorizing the classification, session and protocols handling. MiddleClick could factorize some functionalities to remove redundant code. However the observations of chapter 5 suggest we factorized most of the redundant operations. *E.g.*, it is unlikely that multiple DPIs would be chained.

In a sense, implementing MiddleClick in the OS itself, providing a new kind of socket interface would have been a more logical choice. It would have lead to a tailored Operating System that uses only the parts of the networking stack that are currently asked by the application and that would not loose the concept of service chain by receiving and sending raw packets blindly back and forth between applications. We did not do so because of the problems exposed in chapter 2, leading to the OS being slower even for simple I/O than implementing the software in user space using frameworks like DPDK. Hence, we did not want to build our system on a bad foundation. Fixing Linux limitations as proposed in section 2.4, and adding MiddleClick context-based stack, along with the unified flow and session classification as part of something similar than Linux's NFQ to define the service chain, would certainly be an interesting future work.

We then studied how to use upfront classification hardware to offload the unified classification of MiddleClick, so it can receive packets pre-classified and directly recover the right session state with a minimal classification step. We looked at how to combine multiple NFV instances on the same box without requiring a software classification by exposing the NIC capabilities to a controller that can tag packets. Packets can then be received directly by the right core of the right NFV agent that serves the corresponding service chain. Combined, those two approaches would allow to keep isolation between service chains without any cost, push firewall and most classification task in SDN switches, drop the traffic before it even hits the box and remove classification even inside the NFV server itself. As a future work, we would like to study how the dynamic session classification of MiddleClick could be accelerated using SmartNICs and programmable switches, possibly supporting the P4[105] language. Packets could arrive to the NFV server not only tagged according to their traffic class and the service chain they must follow, but also in a way that will accelerate the identification of their session.

Surprisingly, measuring the performance of high-speed systems have been a badly covered area of research. Reproducibility is a known problem in our field of research, with little to no experiment management system. Indeed, in high-speed networking environments tightly tied to hardware facilities, simulation or virtualization are not a solution. We built a new python-based experiment manager automation tool for high-speed networking, NPF. The tool allows to simply define an experiment in a single file using scripting languages. NPF will deploy multiple software over a cluster, and run the experiment multiple times to ensure stability of the results. The experiment file can define many parameters to grid-search and run again the experiment to find the combination that will lead to the best performance. NPF supports multiple packet generation methods and generates multiple measurements at once such as throughput or latency. The tool automatically build comprehensive graphs and statistical outcomes.

The "final" step of this layered-approach to design the efficient and resilient infrastructure of future networks is in three parts.

Firstly, use the liberated CPU resources to build new efficient functions. MiddleClick allows to easily change any flow on the fly, modify requests to re-route them, or part of them to fast caches. We already started this kind of work while working in the SuperFluidity EU H2020 project related to 5G mobility, re-routing some mobile requests to serve them in the edge. While for consistency this document focuses on datacenter usage, ISPs and particularly mobile ISPs would also benefit from most of these developments. There are a number of potential uses for a better, "fluid", modular and distributed NFV platform.

Secondly, we want to enhance the NFV agents management, improving network distribution techniques. How to allocate enough resources to the service chains to ensure a certain SLA (latency bound, throughput, ...) ? How to scale the service chain among multiple servers ? While we introduce bits of ideas for cooperation with an NFV controller, the topic of VNF migration, and how to use horizontal scaling (using multiple NFV agents in parallel) to distribute the work among multiple servers is still uncovered and is one of our major future works. We'll have to look at "service chain" scheduling and placement to ensure bounded latency under maximized throughput. This starts with the characterization of building blocks. That is, given an amount of resources, what will be the performance of those blocks? One way to do it is to use an evolved NPF to characterize the performance of the blocks. Using machine learning, one could allow to query the "parameters" (number of cpus, amount of memory, table sizes, . . .) that will be needed to fulfill a given SLA. How to move the learning to a datacenter scale? How to use online learning to use the information gathered from running appliances? Then maybe use reinforcement learning to decide some action about scaling and learn about their impact on the performance.

Thirdly, while we focused on pass-through traffic, some middleboxes actually require connection termination or initiation. Proxy caches terminate some of the connection to serve some files by themselves. While MiddleClick actually supports some client and server semantics, they are not discussed here because they are still in an early stage. We would like to study further the possibility of integrating a full TCP stack, which would follow the idea of factorization and only invoke the minimal features of MiddleClick. Network stack specialization has already be proven to enhance performance[55].

The context of our work is a world of VNFs that can move around, in datacenters to consolidate processing or at the contrary on processing units as close as possible to the final user, *e.g.* in an ISP's backhaul to minimize the latency. The commercialization of virtual middleboxes is an emerging new industry, allowing network operators to outsource their costly and ageing network equipment. Cloud-based NFV allows to activate security features for their network like we download mobile applications today. This will push innovation in the network that ossified in past years while it needs to evolve, and enable innovation through new protocols that could be installed in one click. That will also lead to customized, sometimes per-client service chains that can grow very long. It is our belief that our work covered low-level details of how to build an NFV dataplane agent that can accommodate tomorrow's network using its very *profound* flexibility, that can take advantage of hardware offloading capabilities as hardware evolves, but already very efficient in pure software implementation.

Appendices

Appendix A

More results about distributed packet processing

A.1 Increasing proportion of access to the packet content

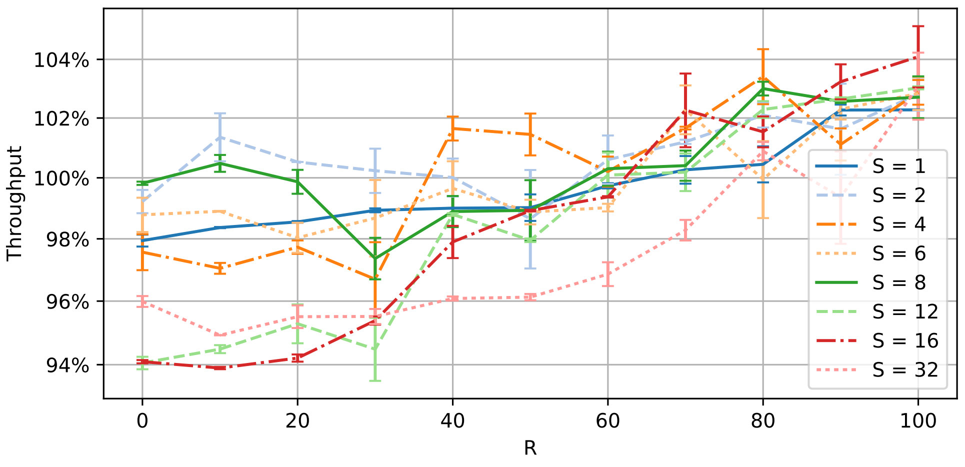


Figure A.1: Relative improvement of the parallel approach over the returning pipeline approach using **two** processing stages running on **two** cores under an increasing percentage of access to the packet data instead of the array. $N = 100$

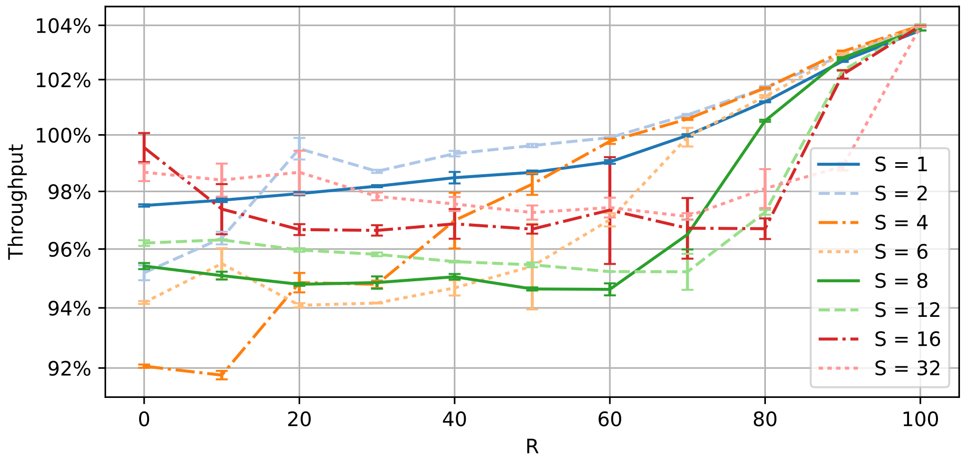


Figure A.2: Relative improvement of the parallel approach over the returning pipeline approach using **eight** processing stages running on **eight** cores under an increasing percentage of access to the packet data instead of the array. $N = 100$

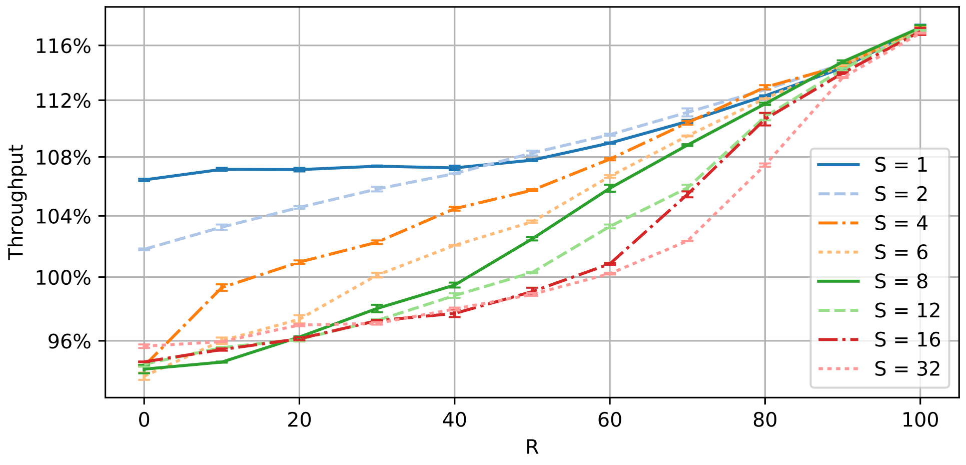


Figure A.3: Relative improvement of the parallel approach over the returning pipeline approach using **four** processing stages running on **four** cores under an increasing percentage of access to the packet data instead of the array. $N = 20$

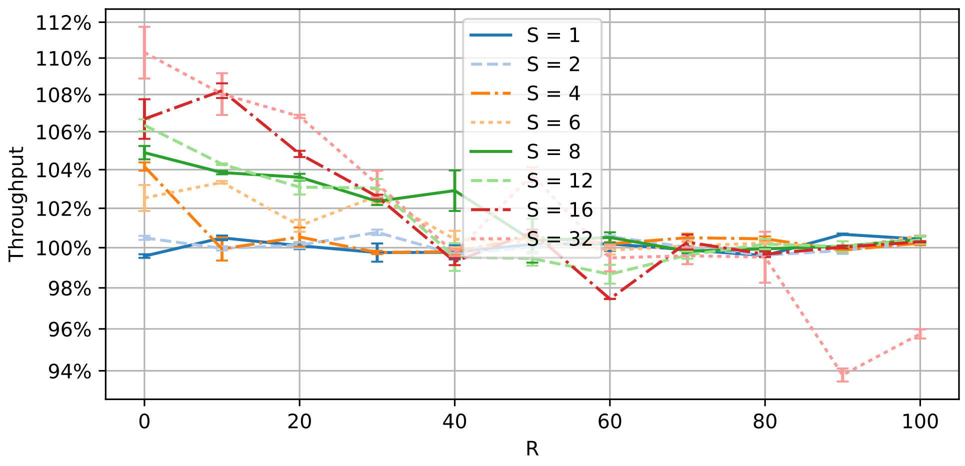


Figure A.4: Relative improvement of the parallel approach over the returning pipeline approach using **four** processing stages running on **four** cores under an increasing percentage of access to the packet data instead of the array. $N = 2000$

A.2 Increasing number of cores

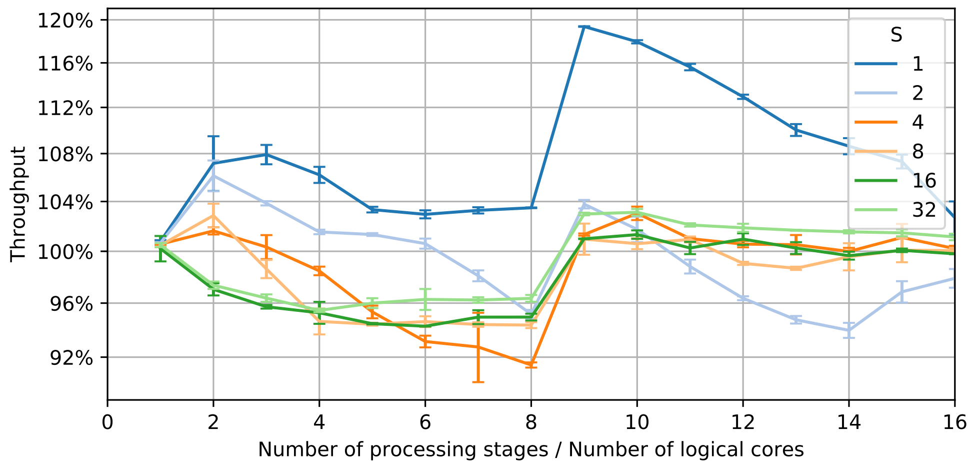


Figure A.5: Relative improvement of the parallel approach over the returning pipeline approach doing $N = 20$ memory access to an array of size S for an increasing number of processing stages. One logical core is used per processing stages. Cores 9 to 16 are hyper-threads.

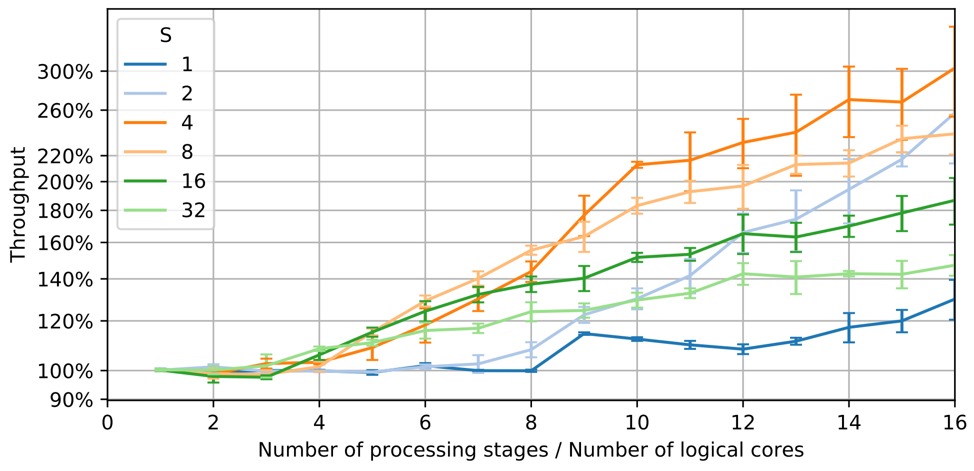


Figure A.6: Relative improvement of the parallel approach over the returning pipeline approach doing $N = 2000$ memory access to an array of size S for an increasing number of processing stages. One logical core is used per processing stages. Cores 9 to 16 are hyper-threads.

Appendix B

More results about userlevel parallel data structures

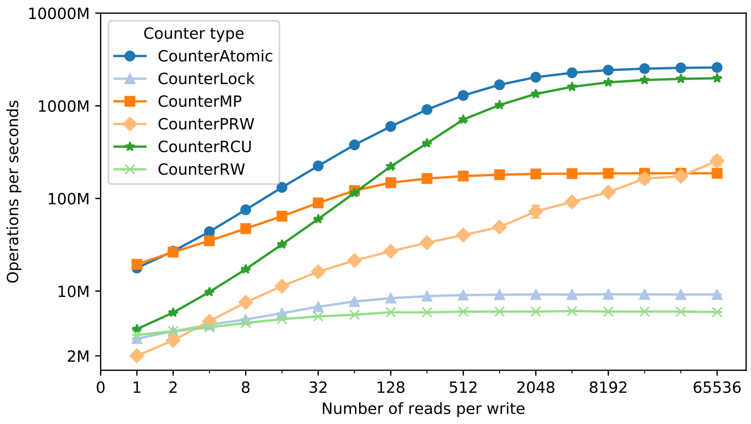


Figure B.1: Number of reads (read 64 bits count and bytes count) or write (add packet count and number of bytes to the counter) per seconds with increasing read rate. Using 8 cores from a single processor.

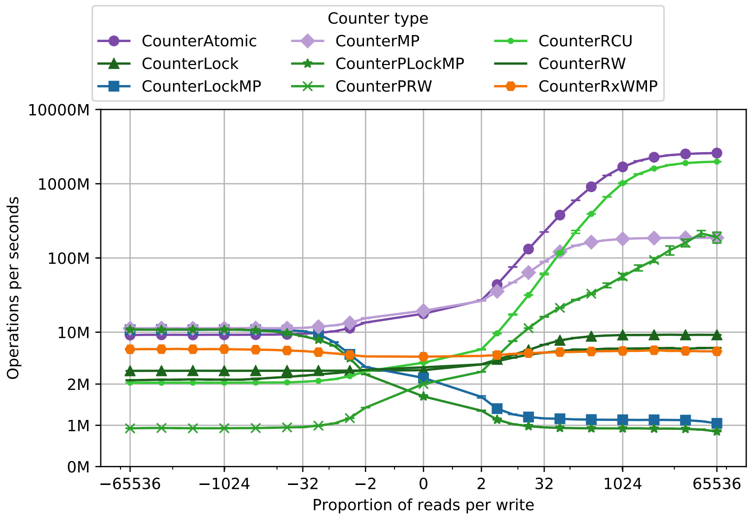


Figure B.2: Performance of data structures of interest in terms of operations per seconds under a various amount of reads or writes. On the left, the graph shows the performance of the structures in a write-mostly situation. A value of -65536 means 65536 writes per read. This value decrease up to 0, meaning that read and writes are on a par. While on the opposite when the value reaches 65536, 65536 reads are executed per write. Using 8 cores from a signle processor. Both axes use a log scale. Note that CounterAtomic and CounterMP do not allow consistent read of the structure, while CounterLockMP is not doing fresh reads.

B.0.1 RxWMP data structure with read or write preference

The RxWMP method can be modified to enable a preference for reads or for write operations. Having a preferred operation allows to bound the amount of time the preferred operation will wait before accessing the structure. The RxW lock with a preference for read or write is actually a solution to the classical shared shower problem of multi-core programming. For ease of comprehension, we will focus on the preferred-read solution, the preferred-write only needing to reverse the logic. Men and women can access the showers but never at the same time. When men are in the shower, other men can join and the same applies to women. When the showers are accessed by men, but a woman wants to access, no further man can enter the shower to ensure that the showers are available for women as soon as possible. This applies for readers in a preferred-read situation as we want the reader to wait for a bounded time before accessing the per-thread structure, as writers (men in the metaphor) may come one by one always keeping the lock (the shower) busy without letting any chance for a reader to grab it (a woman to enter the shower).

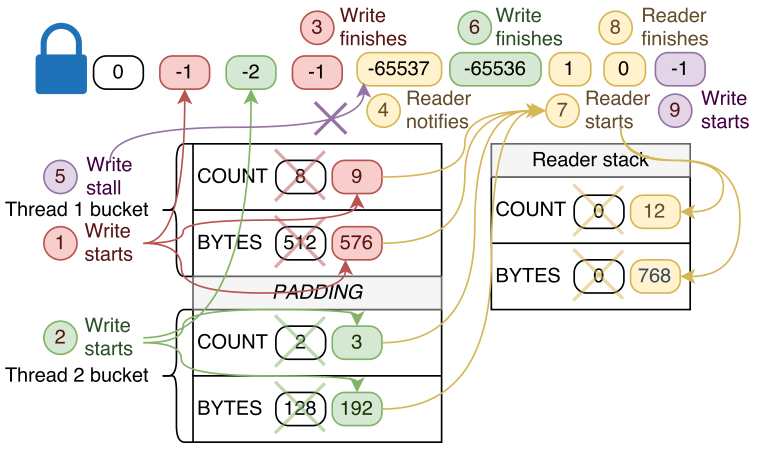


Figure B.3: Per-thread duplication approach protected per an RxW lock (preferred read) to allow for consistency. The numbers in circles show the sequence of events.

Pseudocode for the read acquire sequence is algorithm 4 and the write sequence in algorithm 5. In the context of write mostly the readers may wait for too long before accessing the variable, therefore if a reader cannot lock directly (because the value is strictly lesser than 0 indicating some writers are accessing the structure), it will prevent further writers from taking the lock by adding -MAX_WRITER to the value using a CAS. MAX_WRITER is an upper bound on the maximal number of concurrent writers. Before operating the CAS instruction, the writers check if the value is not lower than -MAX_WRITER and wait for it to become bigger again meaning that the reader has accessed the data and actually finished reading.

Figure B.3 shows a sequence of action where two threads access the bucket to update their counter value (the lock becomes -1 then -2 as shown per the sequence 1 and 2). In this example, MAX_WRITER is arbitrarily fixed to 65536. One the thread finishes (the value becomes -1 as shown per the event 3). A reader wants to grab the lock, but as there are currently some writers, the lock becomes -65537 in event 4 to prevent further writer to grab the lock. In event 5, a writer arrives but spinloop as the value is lower than -65536. The the first writer finishes, changing the to -65536 in event 6. The reader that subtracted 65536 from the value can then change it to 1, to grab the lock in event 7, compute the sum by traversing the full array until it finishes and put back to 1 the value in event 8. At which points the last writer that was stalled because a reader wanted to access the array can grab the value by setting it to -1 in event 9.

Algorithm 4 RxW Lock read

```
function READ_BEGIN
    while true do
        $current \leftarrow refcnt$
        if $current \geq 0$ then                              ▷ Unlocked or reader presents
            if refcnt.CAS(current, current + 1) then
                break
            end if
        else if $current > -65536$ then                ▷ Writers present, no pending reader
            if refcnt.CAS(current, current - 65536) then ▷ Notify our reader presence
                repeat
                    relax cpu
                until $refcnt == -65536$                        ▷ Wait for all writers to finish
                $refcnt \leftarrow 1$
                break
            end if
        end if
        relax cpu
    end while
end function
function READ_END
    atomic $refcnt - -$
end function
```

Algorithm 5 RxW Lock write

```
function WRITE_BEGIN
    while true do
        $current \leftarrow refcnt$
        if $current \leq 0$ and $current > -65536$ then
            if refcnt.CAS(current, current - 1) then
                break
            end if
        end if
        relax cpu
    end while
end function
function WRITE_END
    atomic $refcnt++$
end function
```

Figure B.4 shows the performance of the preferred-read and preferred-write versions of the RxWMP data structure compared to the agnostic one. While performance do not increase using one or the other, we do believe ensuring one operation completes as soon as possible is useful in certain situations. In the counter example, one might want to aggregate the values on rare occasions, without competing too much with other cores to avoid jitter while doing so.

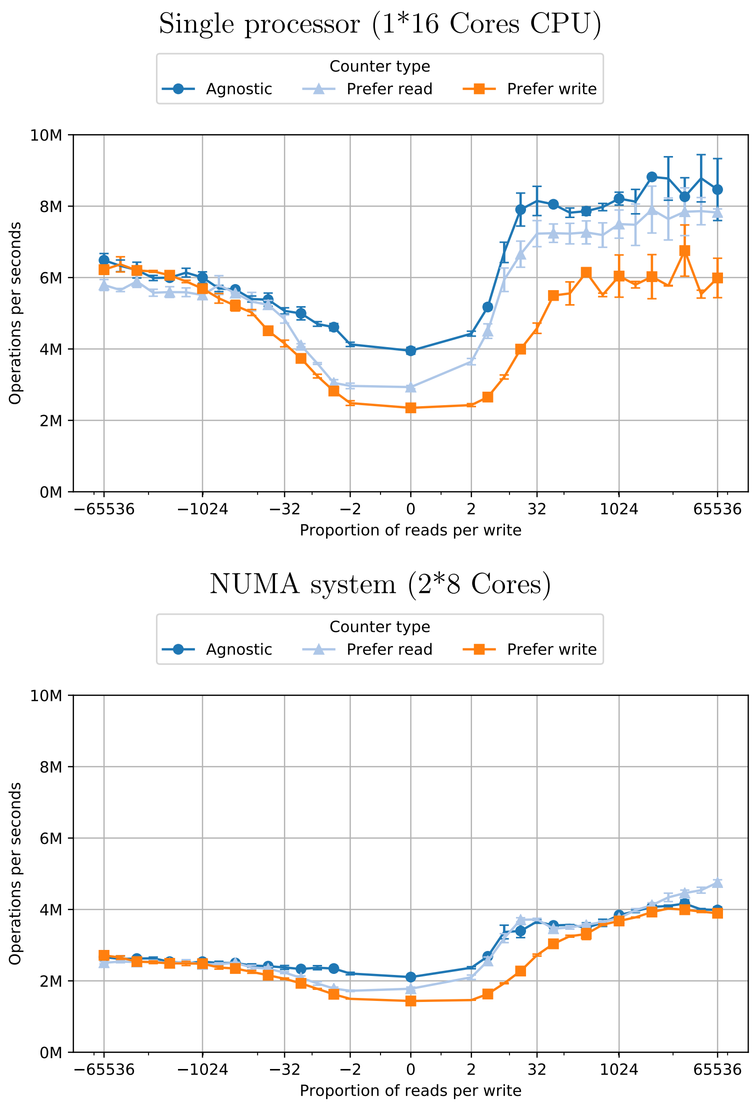


Figure B.4: Performance of the RxWMP data structure with read or write preference

References

[1] B. Carpenter and S. Brim. **Middleboxes: Taxonomy and Issues**. RFC 3234, Internet Engineering Task Force, February 2002. 1, 2

[2] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. **Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service**. In *Proc. ACM SIGCOMM*, August 2012. 2, 3, 12

[3] Rahul Potharaju and Navendu Jain. **Demystifying the dark side of the middle: a field study of middlebox failures in datacenters**. In *Proc. ACM Internet Measurement Conference (IMC)*, October 2013. 2, 3

[4] Gregory Detal, Benjamin Hesmans, Olivier Bonaventure, Yves Vanaubel, and Benoit Donnet. **Revealing middlebox interference with tracebox**. In *Proc. ACM Internet Measurement Conference (IMC)*, October 2013. 3

[5] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. **Is it still possible to extend TCP?** In *Proc. ACM Internet Measurement Conference (IMC)*, November 2011. 3

[6] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. **Are TCP Extensions Middlebox-proof?** In *Proc. Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2013. 4

[7] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. **Fast and Scalable Layer Four Switching**. *Proc. ACM SIGCOMM*, October 1998. 4

[8] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. **Scalable high speed IP routing lookups**. In *Proc. ACM SIGCOMM*, 1997. 4

[9] Pankaj Gupta and Nick McKeown. **Packet classification on multiple fields**. *Proc. ACM SIGCOMM*, 1999. 4

[10] Edward Guillen, Ana María Sossa, and Edith Paola Estupiñán. **Performance Analysis over Software Router vs. Hardware Router: A Practical Approach**. In *Proc. IAENG World Congress on Engineering and Computer Science (WCECS)*, 2012. 4

[11] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. **The Click Modular Router**. *ACM Trans. Comput. Syst.*, **18**(3):263–297, August 2000. 4, 17, 32, 37, 129, 177, 181

[12] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. **The Power of Batching in the Click Modular Router**. In *Proc. ACM Asia-Pacific Workshop on Systems (APSYS)*, 2012. 4, 38, 42, 61

[13] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. **NBA (network balancing act): A high-performance packet processing framework for heterogeneous processors**. In *Proc. ACM European Conference on Computer Systems (EuroSys)*, 2015. 4, 10

[14] Andrea Bianco, Robert Birke, Davide Bolognesi, Jorge M Finochietto, Giulio Galante, Marco Mellia, MLNPP Prashant, and Fabio Neri. **Click vs. Linux: two efficient open-source IP network stacks for software routers**. In *IEEE Workshop on High Performance Switching and Routing (HPSR)*, May 2005. 4

[15] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. **RouteBricks: Exploiting Parallelism to Scale Software Routers**. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, October 2009. 5, 32, 38, 91

[16] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. **Controlling parallelism in a multicore software router**. In *Proc. ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, November 2010. 5, 78, 84

[17] Katerina Argyraki, Salman Baset, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Eddie Kohler, Maziar Manesh, Sergiu Nedevschi, and Sylvia Ratnasamy. **Can Software Routers Scale?** In *Proc. ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, August 2008. 5

[18] Benjie Chen and Robert Morris. **Flexible Control of Parallelism in a Multiprocessor PC Router.** In *Proc. USENIX Annual Technical Conference (ATC)*, June 2001. 5

[19] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. **Fairness issues in software virtual routers**. In *Proc. ACM Workshop on Programmable routers for extensible services of tomorrow (PRESTO)*, August 2008. 5

[20] Yong Liao, Dong Yin, and Lixin Gao. **PdP: Parallelizing Data Plane in Virtual Network Substrate**. In *Proc. ACM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)*, 2009. 5

[21] Tom Barbette. **GitHub - FastClick**, 2015. `https://github.com/tbarbette/fastclick`. 5, 6, 15, 67, 112

[22] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. **Stateless datacenter load-balancing with beamer**. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018. 5

[23] Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. **Profiling and accelerating commodity NFV service chains with SCC**. *Journal of Systems and Software*, **127**:12–27, May 2017. 5

[24] Hassan Jameel Asghar, Luca Melis, Cyril Soldani, Emiliano De Cristofaro, Mohamed Ali Kaafar, and Laurent Mathy. **Splitbox: Toward efficient private network function virtualization**. In *Proc. ACM Workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2016. 5

[25] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. **Metron: NFV Service Chains at the True Speed of the Underlying Hardware**. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018. 5, 10, 168

[26] Leonardo Linguaglossa, Dario Rossi, Dave Barach, Damjan Marjon, and Pierre Pfiester. **High-speed Software Data Plane via Vectorized Packet Processing**. Technical report, Telecom ParisTech, CNIT and University of Rome Tor Vergata, Systems, Inc., 2018. 5

[27] Rafael Laufer, Massimo Gallo, Diego Perino, and Anandatirtha Nandugudi. **CliMB: enabling network function composition with click middleboxes**. *Proc. ACM SIGCOMM*, 2016. 5, 8, 117, 129

[28] Davide Kirchner, Raihana Ferdous, Renato Lo Cigno, Leonardo Maccari, Massimo Gallo, Diego Perino, and Lorenzo Saino. **Augustus: a CCN router for programmable networks**. In *Proc. ACM Information-Centric Networking (ICN)*, 2016. 5

[29] Sebastian Gallenmüller, Paul Emmerich, Rainer Schönberger, Daniel Raumer, and Georg Carle. **Building Fast but Flexible Software Routers**. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2017. 5

[30] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, Laurent Mathy, and Panagiotis Papadimitriou. **Forwarding path architectures for multicore software routers**. In *Proc. of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, November 2010. 5

[31] Linux Foundation. **Data Plane Development Kit (DPDK)**, 2015. `http://www.dpdk.org`. 5, 17, 24, 25, 37, 117, 123, 146, 178, 189

[32] Luigi Rizzo. **netmap: A Novel Framework for Fast Packet I/O.** In *Proc. USENIX Annual Technical Conference (ATC)*, 2012. 5, 17, 24, 25, 28, 37, 117, 120, 146, 178, 189

[33] ntop. **PF_RING**. `http://www.ntop.org/products/pf_ring/`. 5, 24, 25, 146

[34] Solarflare. **OpenOnload**. `http://www.openonload.org/`. 5, 24, 26, 129

[35] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. **PacketShader: A GPU-accelerated Software Router**. In *Proc. ACM SIGCOMM*, August 2010. 5, 10, 24, 28, 38, 146

[36] Björn Töpel. **Introducing AF_XDP support**. 5

[37] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. **Arrakis: The Operating System is the Control Plane**. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014. 6, 24, 68, 117, 128

[38] James W Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. **xOMB: extensible open middleboxes with commodity servers**. In *Proc. ACM/IEEE symposium on Architectures for networking and communications systems (ANCS)*, 2012. 7, 129, 130

[39] Georgios P Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q Maguire Jr, and Dejan Kostić. **SNF: synthesizing high performance NFV service chains**. *PeerJ Computer Science*, **2**:e98, 2016. 7, 8, 118, 131

[40] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. **Design and implementation of a consolidated middlebox architecture**. In *Proc. USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012. 7, 118, 129

[41] Anat Bremler-Barr, Yotam Harchol, and David Hay. **OpenBox: a software-defined framework for developing, deploying, and managing network functions**. In *Proc. ACM SIGCOMM*, 2016. 8, 68, 117, 130, 172

[42] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. **NetBricks: Taking the V out of NFV.** In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. 8, 128, 131, 169

[43] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. **E2: a framework for NFV applications**. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2015. 8, 68, 159, 165, 170, 172

[44] Mehdi Bezahaf, Abdul Alim, and Laurent Mathy. **FlowOS: A Flow-based Platform for Middleboxes**. In *Proc. ACM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2013. 8, 156

[45] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. **mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.** In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. 8, 117, 120, 129

[46] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. **mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes.** In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017. 8, 117, 128, 131, 144, 159

[47] Yunhong Gu and Robert L Grossman. **UDT: UDP-based data transfer for high-speed wide area networks**. *Computer Networks*, **51**(7):1777–1799, 2007. 9

[48] Tom Barbette. **GitHub - MiddleClick**, 2018. `https://github.com/tbarbette/fastclick/tree/middleclick`. 10, 15, 163

[49] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. **Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware**. In *Proc. ACM SIGCOMM*, 2016. 10

[50] B Zaluški, B Rajtar, H Habjanić, M Baranek, N Šlibar, R Petračić, and T Sukser. **Terastream implementation of all IP new architecture**. In *Proc. IEEE International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2013. 11

[51] Niv Buchbinder, Navendu Jain, and Ishai Menache. **Online job-migration for reducing the electricity bill in the cloud**. *NETWORKING 2011*, pages 172–185, 2011. 11

[52] Tom Barbette. **NPF**, 2017. `https://github.com/tbarbette/npf`. 15, 179, 187

[53] Wenji Wu, Matt Crawford, and Mark Bowden. **The performance analysis of Linux networking–packet receiving**. *Computer Communications*, **30**(5):1044–1057, 2007. 20

[54] Tom Barbette, Cyril Soldani, and Laurent Mathy. **Fast userspace packet processing**. In *Proc. ACM/IEEE Symposium on Architectures for networking and communications systems (ANCS)*, May 2015. 20, 39, 120, 128, 181

[55] Ilias Marinos, Robert NM Watson, and Mark Handley. **Network stack specialization for performance**. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2013. 21, 117, 129, 193

[56] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. **Speeding up packet I/O in virtual machines**. In *Proc. ACM/IEEE symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013. 22

[57] K Salah and A Kahtani. **Performance evaluation comparison of Snort NIDS under Linux and Windows Server**. *Journal of Network and Computer Applications*, **33**(1):6–15, 2010. 22

[58] Linux Foundation. **Packet_mmap**. https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt. 24

[59] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. **IX: A Protected Dataplane Operating System for High Throughput and Low Latency**. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014. 24, 68, 117, 128

[60] ntop. **DNA vs netmap**. http://www.ntop.org/pf_ring/dna-vs-netmap/. 25

[61] Intel. **Core^{TM} i7-4930K Processor (12M Cache, up to 3.90 GHz)**. http://ark.intel.com/products/77780. 28

[62] ASUS. **P9X79-E WS**. http://www.asus.com/Motherboards/P9X79E_WS/. 28

[63] Tom Barbette. **GitHub - Tilera Packet Generator**. https://github.com/tbarbette/tilerapktgen. 28

[64] Tom Barbette. **GitHub - KForward**, 2017. https://github.com/tbarbette/kforward. 30

[65] Luigi Rizzo. **Device polling support for FreeBSD**. In *BSDConEurope Conference*, 2001. 31

[66] Tom Barbette. **GitHub - A proof of concept for the ability to forward backpressure to the driver in Linux**. https://github.com/tbarbette/linux-backpressure. 33

[67] Tom Barbette. **GitHub - Click branch supporting RX_HANDLER_DROPPED**. https://github.com/tbarbette/linux/tree/kpressure. 33

[68] Tom Barbette. **XDP for Kernel by-pass?**, 2017. https://www.spinics.net/lists/xdp-newbies/msg00247.html. 34

[69] John Fastabend. **[RFC,1/2] af_packet: direct dma for packet ineterface**, 2017. http://patchwork.ozlabs.org/patch/720937/. 35

[70] Weibin Sun and Robert Ricci. **Fast and Flexible: Parallel Packet Processing with GPUs and Click**. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, October 2013. 38, 57, 61

[71] Tom Barbette. **Click pull request #162 to enable multi-producer single-consumer mode in linuxmodule FromDevice**. https://github.com/kohler/click/pull/162. 42

[72] T. Mori, R. Kawahara, S. Naito, and S. Goto. **On the characteristics of Internet traffic variability: spikes and elephants**. In *Proc. IEEE/IPSJ International Symposium on Applications and the Internet (SAINT)*, 2004. 44

[73] fd.io. **Vector Packet Processing (VPP)**. 68

[74] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. **NetVM: high performance and flexible networking using virtualization on commodity platforms**. *IEEE Transactions on Network and Service Management*, **12**(1):34–47, 2015. 68, 128

[75] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. **Meltdown**. *arXiv preprint arXiv:1801.01207*, 2018. 68

[76] Irina Fedotova, Eduard Siemens, and Hao Hu. **A high-precision time handling library**. *J. Commun. Comput*, **10**:1076–1086, 2013. 69, 70

[77] Tom Barbette. **ixgbe: support for ethtool set_rxfh**. `https://github.com/torvalds/linux/commit/1c7cf0784e4d448ed8a07c5fc1e3aac1528272f1`. 89

[78] Linux Foundation. **Userspace RCU Library : What Linear Multiprocessor Scalability Means for Your Application**. `https://linuxplumbersconf.org/2009/slides/Mathieu-Desnoyers-talk-lpc2009.pdf`. 102

[79] **Userspace RCU library**. `http://liburcu.org/`. 103

[80] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. **User-level implementations of read-copy update**. *IEEE Transactions on Parallel and Distributed Systems*, **23**(2):375–382, 2012. 103

[81] P McKenney. **Deterministic synchronization in multicore systems: the role of RCU**. *Aug*, **18**:1–9, 2009. 104

[82] Keir Fraser. **Practical lock-freedom**. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004. 104

[83] Cisco. **Snort - Network Intrusion Detection & Prevention System**, 2017. `http://www.snort.org/`. 113, 120

[84] Willy Tarreau. **HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer**, 2017. `http://www.haproxy.org/`. 113, 120, 161

[85] NGINX Inc. **NGINX | High Performance Load Balancer, Web Server & Reverse Proxy**, 2017. `https://www.nginx.com/`. 113, 120, 161

[86] Luigi Rizzo and Giuseppe Lettieri. **VALE, a Switched Ethernet for Virtual Machines**. In *Proc. ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012. 115

[87] Mauricio Vasquez Bernal, Ivano Cerrato, Fulvio Risso, and David Verbeiren. **Transparent Optimization of Inter-Virtual Network Function Communication in Open vSwitch**. In *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*, pages 76–82. IEEE, 2016. 115

[88] Tom Barbette, Cyril Soldani, Romain Gaillard, and Laurent Mathy. **Building a chain of high-speed VNFs in no time**. In *Proc. IEEE High Performance Switching and Routing (HPSR)*, 2018. Invited Paper, **to appear**. 120

[89] Gaillard Romain. **Lightweight Middlebox TCP**. *Master Thesis*, 2016. 120

[90] Open Information Security Foundation. **Suricata | Open source IDS / IPS / NSM engine**, 2017. `https://suricata-ids.org/`. 120

[91] Linux Foundation. **Linux Perf**. `https://perf.wiki.kernel.org`. 120

[92] **The Squid Proxy Cache**. `http://www.squid-cache.org/`. 123

[93] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. **Unikernels: Library Operating Systems for the Cloud**. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013. 128

[94] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. **ClickOS and the art of network function virtualization**. In *Proc. USENIX Networked Systems Design and Implementation (NSDI)*, April 2014. 128

[95] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. **OpenNetVM: a platform for high performance network service chains**. In *Proc. ACM workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2016. 128

[96] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, and Timothy Wood. **Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization**. In *Proc. ACM International on Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2016. 129

[97] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. **NFP: Enabling Network Function Parallelism in NFV**. In *Proc. ACM Special Interest Group on Data Communication*. ACM, 2017. 129

[98] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. **SIMPLE-fying middlebox policy enforcement using SDN**. *Proc. ACM SIGCOMM*, 2013. 130

[99] Aaron Gember, Robert Grandl, Junaid Khalid, and Aditya Akella. **Design and implementation of a framework for software-defined middlebox networking**. *Proc. ACM SIGCOMM*, 2013. 130

[100] David A Maltz and Pravin Bhagwat. **TCP Splice for application layer proxy performance**. *Journal of High Speed Networks*, **8**(3):225–240, 1999. 131

[101] David Dolson, Matthew Desmond, and Jim Kuhn. **TCP proxy providing application layer modifications**, October 2 2007. US Patent 7,277,963. 131

[102] Shinae Woo and KyoungSoo Park. **Scalable TCP session monitoring with symmetric receive-side scaling**. *KAIST, Daejeon, Korea, Tech. Rep*, 2012. 144

[103] Netronome. **Netronome Smart NICs**. `https://www.netronome.com/products/smartnic/overview/`. 145, 166

[104] Mellanox. **Mellanox Ethernet Cards**. `http://www.mellanox.com/page/ethernet_cards_overview`. 145, 166

[105] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. **P4: Programming protocol-independent packet processors**. *Proc. ACM SIGCOMM*, 2014. 148, 191

[106] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. **Packet classification using multidimensional cutting**. In *Proc. ACM SIGCOMM*, 2003. 151

[107] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. **SoftNIC: A software NIC to augment hardware**. *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155*, 2015. 159, 165

[108] Will Glozer. **WRK**. `https://github.com/wg/wrk`. 159, 187

[109] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. **High performance network virtualization with SR-IOV**. *Journal of Parallel and Distributed Computing*, **72**(11):1471–1480, 2012. 165

[110] Jiuxing Liu. **Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support**. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010. 165

[111] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. **DevoFlow: Scaling Flow Management for High-performance Networks**. In *Proc. ACM SIGCOMM*, August 2011. 167

[112] Voravit Tanyingyong, Markus Hidell, and Peter Sjödin. **Using hardware classification to improve pc-based openflow switching**. In *Proc. IEEE High Performance Switching and Routing (HPSR)*, 2011. 168

[113] Eric Eide. **Toward replayable research in networking and systems**. *Position paper presented at Archive*, 2010. 176

[114] **ACM SIGCOMM 2017 Reproducibility Workshop (Reproducibility'17)**. `http://conferences.sigcomm.org/sigcomm/2017/workshop-reproducibility.html`. 176

[115] Quirin Scheitle, Matthias Wählisch, Oliver Gasser, Thomas C Schmidt, and Georg Carle. **Towards an ecosystem for reproducible research in computer networking**. In *Proc. ACM Reproducibility Workshop (Reproducibility)*, August 2017. 176

[116] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. **Large-scale Virtualization in the Emulab Network Testbed.** In *Proc. USENIX Annual Technical Conference (ATC)*, 2008. 178

[117] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. **Planetlab: an overlay testbed for broad-coverage services**. *Proc. ACM SIGCOMM*, 2003. 178

[118] Mininet Team. **Mininet**, 2014. 178

[119] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. **Reproducible network experiments using container-based emulation**. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012. 178

[120] Maurizio Pizzonia and Massimo Rimondini. **Netkit: easy emulation of complex networks on inexpensive hardware**. In *Proc. International Conference on Testbeds and research infrastructures for the development of networks & communities (TRIDENTCOM)*, 2008. 178

[121] Kevin Fall and Kannan Varadhan. **The network simulator (ns-2)**. *URL: http://www.isi.edu/nsnam/ns*, 2007. 178

[122] Alina Quereilhac, Mathieu Lacage, Claudio Freire, Thierry Turletti, and Walid Dabbous. **NEPI: An integration framework for network experimentation**. In *Proc. IEEE Software, Telecommunications and Computer Networks (SoftCOM)*, 2011. 178

[123] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. **Moongen: A scriptable high-speed packet generator**. In *Proc. of the ACM Internet Measurement Conference (IMC)*, 2015. 182

[124] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. **Scikit-learn: Machine Learning in Python**. *Journal of Machine Learning Research*, **12**:2825–2830, 2011. 186

[125] Apache. **Apache Benchmark**. https://httpd.apache.org/docs/2.4/programs/ab.html. 187

[126] **IPerf**. https://iperf.fr/. 187

[127] HewlettPackard. **Netperf**. https://hewlettpackard.github.io/netperf/. 187