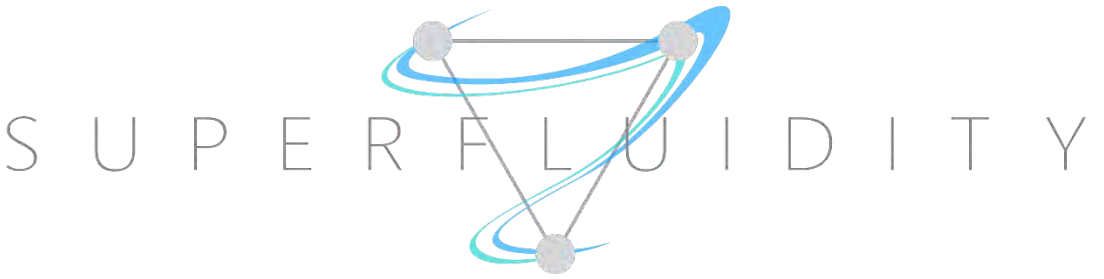


SUPERFLUIDITY

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

Research and Innovation Action GA 671566

DELIVERABLE D5.1:
FUNCTION ALLOCATION ALGORITHMS IMPLEMENTATION AND EVALUATION

Deliverable Type:	Report
Dissemination Level:	PU
Contractual Date of Delivery to the EU:	M28 (30/10/2017)
Actual Date of Delivery to the EU:	M33 (03/05/2018)
Workpackage Contributing to the Deliverable:	WP5

Editor(s):	ULG
Author(s):	**ULG (Cyril Soldani, Tom Barbette, Heng Pan, Laurent Mathy), Intel (Vincenzo Riccobene, Michael Mcgrath), OnApp (Michal Belczyk, Michail Flouris, Anastassios Nanos, Julian Chesterfield, John Thomson, Kevin Du), TUD (Emil Matus)**
Internal Reviewer(s)	Nok-Fr (Bessem Sayadi), Citrix (Christos Tselios)

Abstract: This document discusses the mapping of network services to hardware resources, using SLA metrics and workload characterization, both for generic cloud platforms, and specific CRAN environments.

Keyword List: Optimization, machine learning, resource allocation, performance

INDEX


List of Figures

List of Tables

Glossary

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
API	Application Programming Interface
ARP	Address Resolution Protocol
ATA	AT (Advanced Technology) Attachment
BF	Bloom Filter
BSD	Berkeley Software Distribution
CLT	Coron Lepoint Tibouchi
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DMA	Direct Memory Access
DPDK	Intel’s Data Plane Development Kit
FIB	Forward Information Base
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GBRT	Gradient Boost Regression Trees
GPL	GNU General Public License
GPU	Graphics Processing Unit
GSO	Generic Segmentation Offload
I/O	Input / Output
ICMP	Internet Control Message Protocol
IOV	I/O Virtualization
IP	Internet Protocol
IRQ	Interrupt ReQuest
KPI	Key Performance Indicator
MAUT	Multi-Attribute Utility Theory
MB	MiddleBox or MegaByte
MPSoC	Multi-Processor SoC
MQ	Multi-Queue
NAPI	New API (see note 3 on page 35)
NAT	Network Address Translation

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NIC	Network Interface Controller
NoC	Network-on-Chip
NPF	Network Performance Framework
NUMA	Non-Uniform Memory Access
PCAP	Packet CAPture
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PEKS	Public-key Encryption with Keyword Search
PNFV	Private NFV
PSIO	PacketShader I/O
QoS	Quality of Service
RAM	Random Access Memory
RAN	Radio Access Network
RSS	Receiver-Side Scaling
RX	Receive
SDM	Space-Division Multiplexing
SDN	Software Defined Network
SFP	Small Form-factor Pluggable
SLA	Service Level Agreement
SoC	System-on-Chip
SR-IOV	Single-Root IOV
TCP	Transmission Control Protocol
TDM	Time-Division Multiplexing
TSO	TCP Segmentation Offloading
TTL	Time-To-Live field of IP header
TX	Transmit

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
UDP	User Datagram Protocol
vCPU	Virtual CPU
VM	Virtual Machine
ZC	Zero Copy

Table 1: SUPERFLUIDITY Dictionary.


1 Introduction

1.1 Deliverable Rationale

This deliverable is the final advancement report about the task 5.1, *Function allocation algorithms implementation and evaluation* of the Superfluidity project.

1.2 Executive summary

Task 5.1 leverages the block and functional abstractions developed in Tasks 4.2 and 4.3 to match a set of network services (decomposed into functional abstractions) to the underlying, available block hardware abstractions. This task identifies SLA (Service Level Agreement) metrics (*e.g.* throughput, delay, power consumption, *etc.*) and manages their mapping onto platform resources (described with parameters, configuration or source files, *etc.*). The problem solved by this task is thus the optimization problem of minimizing the platform resource usage while meeting individual SLAs, in order to automatically derive the best performance out of the platform.

First, section 2 introduces utility functions, which can be used to measure how well a resource allocation is doing according to both business (*e.g.* SLAs) and infrastructure KPIs. This gives us a flexible objective function to be used as an optimization target.

Solving the allocation problem that maps an SLA onto a virtualized execution platform is a high-level endeavor. Before this problem can be tackled appropriately, the issues related to performance impact of traffic processing module placement within hardware components must be studied and understood. Indeed, previous work [43] has shown that slight changes in processing pipeline configurations can result in vast swings in observed forwarding performance for network appliances.

We therefore first seek to characterize the performance of network traffic processing pipelines within stand-alone hardware processing units. To this end, we have analyzed various packet I/O frameworks, polling strategies, pipeline execution models (task-to-core allocation strategies), allocation of networking hardware resources, and so on. This characterization gives insights on how to allocate networking functions onto a stand-alone machine, and leads to a low level component allocation framework called FastClick, which we describe in section 3.


We also study FastClick's fitness for purpose through the development of a novel network cloud function called SplitBox, a privacy preserving filtering function for outsourced firewalls (section 4). This application shows that FastClick automatic resource allocation algorithms on a stand-alone machine actually makes a good use of available resources, providing good performance for a demanding application, while requiring no advanced knowledge of the hardware platform from the network function implementer.

As packet processing is not enough for many network functions, which act on *flows*, we then extend FastClick to support flow processing in section 5. We also extend FastClick to support heterogenous hardware. Both improvements greatly extend the range of potential uses of FastClick.

Finally, we then tackle the optimisation of resource allocation for network functions in Click-based environments. We devise a joint characterization, modeling and optimisation approach based on machine learning in section 7. The proposed approach is very general and can also be applied to other environments, such as the deployment of VMs, which was also promisingly investigated using this method. As this approach requires a network function performance dataset, we first devise in section 6 a tool, the Network Performance Framework (NPF), to help with such experiments.

Section 8 then discusses some hypervisor optimizations for VM-based environments, which were implemented in MicroVisor.

Then, we turn to a third kind of environment, the C-RAN, for which we present a novel, real-time and very efficient resource allocation algorithm in section 9.

Finally, section 10 concludes the document.


2 Utility functions for optimal resource allocation

2.1 Introduction

By leveraging cloud computing service-models, service providers deploy and maintain multiple resource requests across globally distributed data centres. Due to increasing heterogeneity in infrastructure resources as well as the workloads, operators are facing increasing challenges to deliver rapid and intelligent deployment decisions which will be exacerbated with the rollout of 5G. Service providers will need autonomous orchestrators to **optimally decide between multiple deployment options, in order to select the best-match in terms of compute, network, and storage resources** for instantiating the components of a service request whilst **minimising resource usage, meeting Service Level Agreements** (SLAs) and providing the **required level of performance**. This deployment problem thus involves the orchestrator making decisions involving multiple-competing objectives.

Current commercial and open-source orchestration solutions make pessimistic decisions (over-provisioning of resources) to avoid conflicts and performance issues. Approaches published in the literature have addressed aspects of optimal service deployments by using genetic algorithms [57], stochastic bin-packing methods [58] and multiple heuristics [59]. However, these limit the number of objectives which can be considered and do not study the trade-off between the benefits gained by the resource provider *versus* the service customer.

This research work investigated the use of utility functions to present a common unified mathematical metric that quantifies the allocation of resources, required by the service components of a resource request, mapped to the physical infrastructure layer within a network functional virtualised infrastructure (NFVI). This is done by formulating each of these objectives and studying the 'reward' that is acquired per objective. The higher the reward, the higher utility for that objective with respect to the deployment decision.

Utility functions have been used for negotiation purposes as reported by John Wilkes [60] and Macias *et al.* [61], and been used by several researchers for specifying the customer's preferences, *e.g.* for scheduling in Tetrisched [62], where the authors focus on predicted runtimes and job-preferences, and Jockey [63], with dynamic estimation for maximising job utility. Network Utility Maximisation [64] is another area which looks at improving throughput of a network by maximising utility with trade-offs between rate and reliability [65]

also being evaluated. In contrast to these approaches, our formulation combines sub-utilities of multiple objectives, analysed on a common platform to consistently rank solutions, and thus **capture the benefits and shortcomings of each deployment**.

This approach supports minimisation of platform usage while ensuring that the selected resource allocation and placement option meets the key performance indicator (KPI) requirements for the service. The methodology thus supports quantification of these objectives, presenting a compare and contrast visualisation of possible solutions. This has the added benefit of providing an increased level of intelligence to an orchestrator which potentially enables it to reason over the benefits of one deployment over another, mitigating the abstraction issue in the cloud enabling efficient and performant deployment of 5G services.

2.2 Resource Representations and Mapping

2.2.1 NFVI Landscape

The NFVI Landscape is represented as a graph with multiple resource nodes, which have varying available capacities. These resources can be divided into three distinct layers, namely the physical, virtual and service layers [66]. Each resource node within a layer can be divided into three distinct functional elements namely compute, network and storage as shown in figure 1.

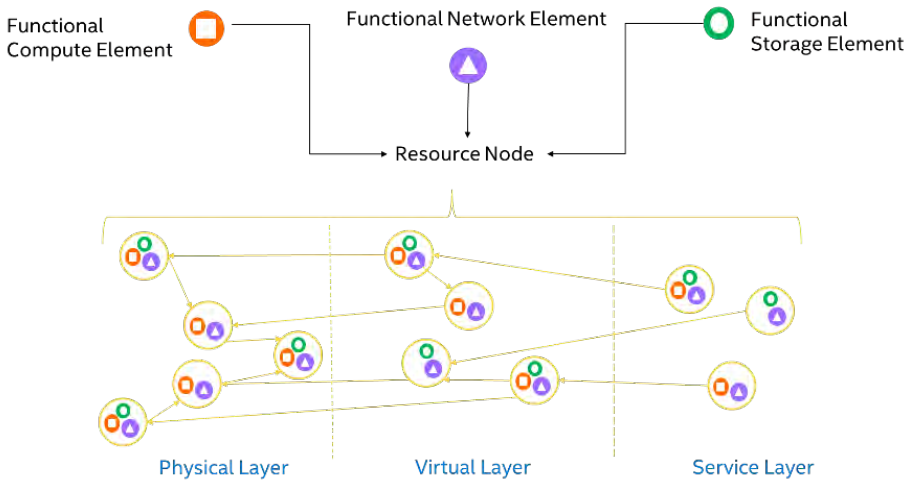


Figure 1: NFVI Landscape Graph.

Each functional element in the resource node $x \in X$ has a capacity value v defining the **platform server components** available depending upon the category of the functional element. Mathematically, this is represented as v_x^c for compute (calculated based on number of cores and RAM of the resource

aggregate), v_x^n for network (calculated based on bandwidth of the enabled network interface card) and v_x^s for storage (calculated based on disk size).

Additionally, several parameters are defined for each node, with subscripts c, n, s to denote the functional elements. These include utilizations $U_x^c \in [0,1]$, $U_x^n \in [0,1]$, $U_x^s \in [0,1]$ and saturations $S_x^c \in [0,1]$, $S_x^n \in [0,1]$, $S_x^s \in [0,1]$ which are normalized for time δt. Other parameters include R_x^c, R_x^n, R_x^s denoting the rate of provisioning capacity per hour, Q_x^c, Q_x^n, Q_x^s denoting the equipment cost per hour incurred by the provider and η_x denoting the maintenance cost per hour incurred by the service-provider. The mean power costs related to powering and cooling per hour is given by $\overline{power}$. A set of H failure-tolerant implementation can be implemented with R_h as the associated revenue.

The edges or communication links are represented by set E. Each edge $e \in E$ is associated with it maximum bandwidth B_e, with rate R_e^b for a unit of bandwidth and the geographical distance covered $len(e)$ which remain constant. The latency of the link l_e and throughput defined using available bandwidth τ_e are normalised for time δt.

2.2.2 Resource Request

The request is also represented in the form of a graph. Each request node in the graph $z \in Z$ represents a service component and the edges in the graph $f \in F$ represent the links between these service components. These nodes are also divided in terms of the terms compute, network and storage functional elements (analogous to those present in the NFVI landscape graph).

The requested compute, network and storage capacity is indicated by $\varphi_z^c, \varphi_z^n, \varphi_z^s$ for each request node z. The bandwidth requested for edge f is represented as b_f and the requested latency is defined as l_f.

2.2.3 Service Level Agreement

The SLA for the service to be deployed is assumed to contain the following terms:

[1] Total running time for the request, represented by T_r hours

[2] Rate for each template $SLA \in SLAs$, defined as R_{SLA}

[3] Unit cost of SLA Violations represented as $j \in J$,

[4] The list of failure-tolerant implementations $h \in H$.

[5] KPI's to be fulfilled by the deployment, *e.g.* latency and throughput.


2.2.4 Mapping Assumptions

The request nodes $z \in Z$ are mapped onto infrastructure nodes $y \in Y \subset X$. Thus $z \to y$ denotes a node z mapped to node y and $Z \cong Y$. The request edge $f \in F$ is mapped to path or communication link $g \in G \subset E$. Also $\forall e \in g$, we can safely say that:

- τ_e is flow on this edge is equal to b_f
- $l_f = l_e$

2.3 Deployment using Multi Attribute Utility Theory

In this scenario, the study and implementation of the utility theory is done to quantify potential deployment solutions and indicate the preference of one over the other, based on the service-provider's objectives. Three attributes are formalised considering service-level metrics and system-level metrics to define these objectives. This supports a unified perspective, by formulating both the context of the resource provider and service customer. These are evaluated based on the Multi-Attribute Utility Theory (MAUT), which supports individual utility definitions for each attribute, determined by understanding the 'reward' that is acquired per attribute, depending on the preferences of the decision-maker. Using this, a multiplicative utility function is formulated, that examines the attributes as non-independent entities. The decision is then to select the deployment solution which allows a balanced trade-off based on these sub-utility functions of the attributes. Our formulation is based on the following three attributes and their associated utility. These are defined and presented in detail later.

- Performance Viability
- Economic Viability
- Service Distribution

The methodology supports ranking of all deployment solutions from the least to most desirable. Figure 2 shows the process flow which was developed.

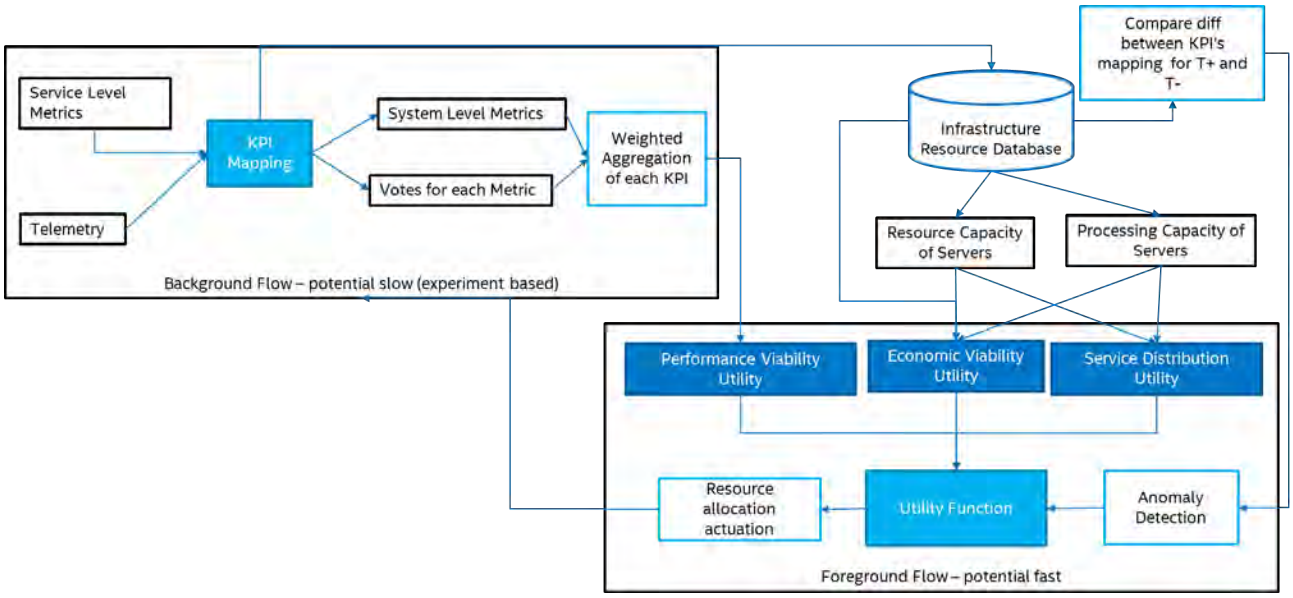


Figure 2: Multi-Attribute Utility Theory workflow.

2.4 Utility Function Definitions

2.4.1 Performance Viability

This utility relates to the ability of the deployment to meet the desired KPI's. This utility helps in **obtaining the best performance from the system**, as it quantifies the QoS.

These performance metrics can be difficult to measure and the KPI Mapping approach (see D4.1 Hardware Selection, Modelling and Profiling) acts like a filtering mechanism for dimensional reduction of collected telemetry data associated with the performance metric. This allows us to compute the performance metrics. Exploiting the existing insights of the KPI mapping work, performance viability can thus be defined in terms of I, C and T since KPI = $F(I,C)_T$ where I is the Infrastructure, C is the configuration and T is the time.

The utility function is created using a weighted aggregation of the top 10 system-level metrics values to which the KPI is mapped. The weights are given by the number of votes[1] attained by each metric, which indicates how strongly this metric influences the KPI. This ensures that the evaluation methodology is generic and can be adapted to different KPI's and mapped system-level metrics.

1 Eight different algorithms are used to identify in the KPI mapping analytics pipeline the relevant features of influence. Features are weighted on basis of the number of votes received, *i.e.* the number of times the metric is identified across the eight algorithms (the max number of votes is thus 8).

Mathematically, the utility function for the performance viability for deployment Z is represented by $U(P_Z)$.

For example, consider a latency KPI which is used as a performance indicator and is mapped to system metrics with current value $i \in I$ with votes v_i present as an output from the KPI Mapping framework.

The utility for this KPI can be denoted by: $U(P_Z) = \sum_i v_i * i$.

2.4.2 Economic Viability

This utility is used to investigate the fulfillment of SLA requirements, in the context of the economical viability of a given deployment. This relates to the profit made by the resource provider whilst **considering all SLA metrics**, including the energy, power and cost of infrastructure. These are embodied as Revenue and Expenditure values. The profit (%) available to the service provider for deployment Z is defined as:

$$p_Z(\%) = \frac{Revenue - Expenditure}{Expenditure} * 100$$

Revenue is composed of the following terms:

- Capacity Revenue = $T_r * \sum_{z \to y \in Y} \varphi_z^c * R_y^c + \varphi_z^n * R_y^n + \varphi_z^s * R_y^s$
- Revenue for bandwidth allocated for edge = $\sum_{g \in G} \sum_{e \in g} \tau_e * R_e^b$
- Revenue from the SLA template = $T_r * \sum_{SLAs} R_{SLA}$

Expenditure is composed of the following terms:

- Equipment Costs = $T_r * \sum_{z \to y \in Y} \varphi_z^c * Q_y^c + \varphi_z^n * Q_y^n + \varphi_z^s * Q_y^s$
- Service and Maintenance Cost = $T_r * \sum_Y \eta_y$
- Power Consumption and Cooling Costs = $T_r * \overline{power} * |Y|$
- SLA Violations (assuming the violation occurs $\bar{j}$ times in an hour) = $T_r * \sum_{j \in J} j * \bar{j}$.

When elucidating the utility function, one needs to consider that a minimum profit level might be set by the service provider, represented by p_{min} which must be attained. Mathematically, the utility function for the economic viability for deployment Z is represented by $U(E_Z)$ and formulated as below:


$$U(E_Z)=\begin{cases}0, & p_Z \le p_{min} \\ \theta_1 * p_Z(\%), & otherwise\end{cases}$$

2.4.3 Service Distribution

This utility function is used to quantify how the resources are distributed over the different servers/nodes. Different rules can be defined for quantifying good service distribution, and in this scenario is related to the consolidation of services. Thus a higher utility indicates that the **minimal number of resources** are used for the deployment, and more capacity is available on the servers after deployment of service Z.

Mathematically, the utility function for the service distribution for deployment Z is represented by $U(S_Z)$ and formulated as follows:

$$U(S_Z)=\frac{|Y|}{\sum_y (v_y^c-\varphi_z^c)+(v_y^n-\varphi_z^n)+(v_y^s-\varphi_z^s)}$$

2.5 Composing a Deployment Utility Function

The utility function for a deployment is calculated by decomposed assessment of the sub-utilities of the attributes. For each of these, α_k indicates weight or a priority value of the attribute while β_k is an additive weight that stores dependence on other attributes.

$$U_Z=(\alpha_1 * U(P_Z)+\beta_1)(\alpha_2 * U(E_Z)+\beta_2)(\alpha_3 * U(S_Z)+\beta_3) \quad \text{s.t.} \quad -\alpha_1+\alpha_2+\alpha_3=1$$

2.6 Experimental Protocol

In order to investigate the application of utility functions to support deployment decision across heterogeneous resource environments an experimental campaign was carried out. There was a particular focus on the effect of system-level metrics on the KPI (*e.g.* correlations performance degradation or improvement).

The high level experimental configuration is shown in figure 3. The NFVI testbed used comprised of the following three compute nodes.

- Intel® Xeon® E5-2699v4 @2.2GHz, 44 Core, 132 GB RAM
- Intel® Xeon® E5-2620 v3 @2.4GHz, 12 Cores, 396 GB RAM
- Intel® Xeon® D-1540 @2.0 GHz, 8 Cores, 32 GB RAM

The video transmuxing service used for the experiments comprised of two Unified Origin instances deployed as VMs and an NGINX HTTP load balancer

also deployed as VMs in an OpenStack Mitaka environment. The service was deployed using a software framework (see D4.1/D6.1 for details) sequentially on each node. A Hammer workload generator ran as a non-virtualised deployment on a dedicated compute node (E5620 @2.4 GHz, 32 GB RAM). Hammer was used for user equipment (UE) simulation and for the measurement of the latency and throughput KPIs. The actions Hammer were controlled by the framework which also collected the KPI metrics (latency and throughput) at the end of each test.

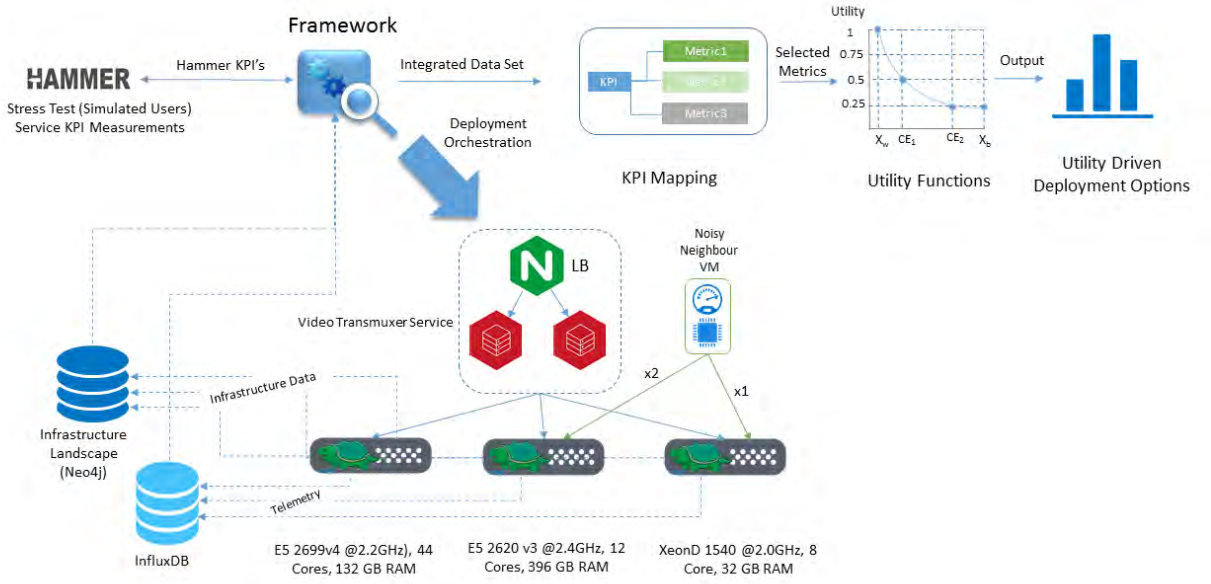


Figure 3: High Level Configuration for Utility Function Experiments.

The behaviour of the compute nodes were modified through the use of noisy neighbour VMs. The noisy VM had an allocation of 2 vCPUs and 4 GB RAM. The VM executed a simple repetitive mathematical application which utilised 100% of the allocated vCPU's which essentially made them unavailable for any other compute operations. One noisy neighbour VM was deployed on the Xeon D node and 2 where deployed on E5 2620 node.

A resource landscaper service was used to capture and store the physical, virtual and service resources and their relationships within the NFVI testbed in a Neo4j graph database. Details on the implementation of the Infrastructure Landscape Service are available in D4.2 - High Performance Block Abstraction Implementations. The performance of the NFVI infrastructure was monitored using the snap telemetry platform with the collected metrics being persisted to an InfluxDB time series database (see D4.1 for additional details).

The framework provided full orchestration of the experimental lifecycle including deployment of the component service VM's, execution of service


stressing via Hammer, aggregation of Hammer and snap telemetry data, execution of automated KPI mapping, data output to utility function. The service was deployed three times on each node. At the end of each deployment cycle the framework generated an integrated data set (KPIs (Latency, Throughput, annotated snap metrics, KPI mapping metrics). The integrated data set was then used for the calculation of the performance, economic and distribution utility for each of the compute nodes.

2.7 Input and Setup of the Utility Calculation Module

The mathematical formulations were adapted to real-world implementation definitions of attributes that were used to study the three utilities defined above, for deploying a Unified Origin service request on three heterogeneous compute hosts on an NFVI infrastructure landscape. The calculation of each individual utility as such required additional inputs and experimental setup and is given below. Furthermore, for each experiment, the aggregate mean value of the utility was computed across the three runs.

2.7.1 Calculation Economic Utility

To quantify the economic utility, two costs models, namely revenue and expenditure were designed based on real world scenarios for the defined infrastructural landscape. For each functional element of the three compute hosts presented above, two dollar costs were defined, the former for dedicating unit capacity to the service component in the revenue model and the latter as an operating expense to the resource provider in the expenditure model, *e.g.* two dollar values were associated with a core in a CPU of the compute host. Additionally, telemetry was gathered to compute the power consumed by the compute hosts.

Furthermore to maintain standard procedures for determining revenue and operational expenses, the costs were amortised over four years.

Next, the capacity requested and used by the service chain components was normalised into the units obtained by the telemetry. These were then provided as input to the utility calculation module.

2.7.2 Calculation Service Distribution Utility

To quantify the service distribution utility, the initial capacity of each functional element of the compute hosts was computed. The compute capacity was calculated as a combination of the number of cores and RAM in each compute


host. The network capacity was calculated based on the capacity of the network interface cards of the compute hosts, and the disk space available on each compute host was used to define the storage capacity.

2.7.3 Calculation of Performance Utility

To quantify Performance Utility, relevant parameter identification were completed, then the relationship between KPI's (latency and throughput) and associated service metrics (filtered from telemetric data) to which these KPI's are mapped was investigated. Finally, the KPI mapping metrics were ordered based on the votes associated with each metric.

The Pearson correlation metric [67] was identified and used to quantify the impact of the mapped platform metric on the given KPI. In the case where the correlation was negative, the computed product of the value of the platform metric and its associated weight was taken to negatively impact the performance utility and deducted from the cumulative utility value. In contrast, when the correlation was positive, it was taken to positively impact the performance utility and the computed product was added to the cumulative utility value.

2.8 Results

The first step in the analysis was the calculation of the performance utility score for the three nodes as shown in figure 4. As expected, the highest performance, *i.e.* E5 2699v4, has the highest utility while the Xeon D has the lowest performance utility.

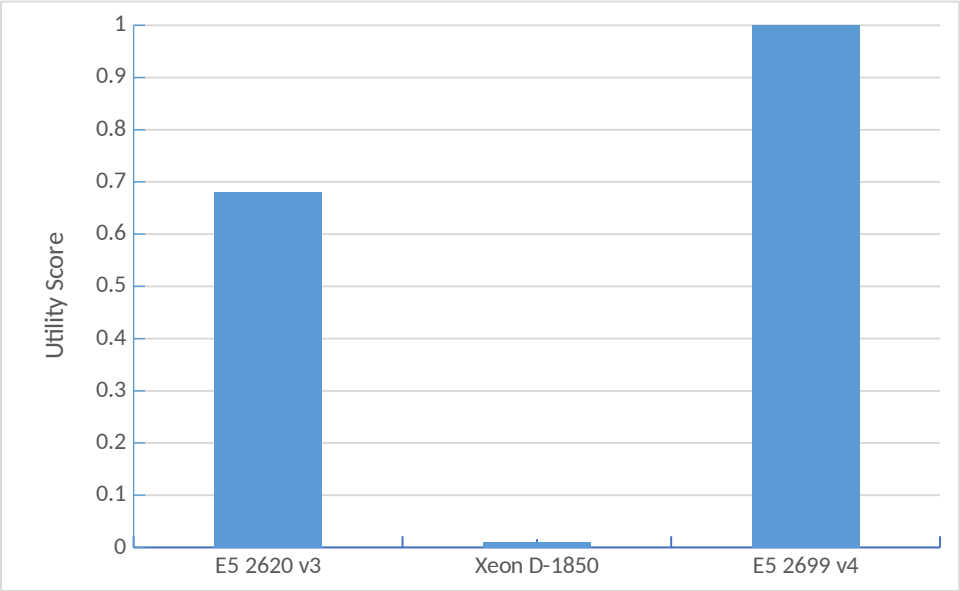


Figure 4: Performance Utility Score across Heterogeneous Compute Nodes.

The economic utility was then calculated across the three compute nodes. The Xeon D was found to have the highest economic utility score, *i.e.* the Xeon D is the cheap node on which to run the service. The Xeon E5 2620 had only a slightly lower economic utility score in comparison to the Xeon D node. As expected the E5 2699 has the lowest economic utility score, *i.e.* running the service on this node incurs the highest cost.

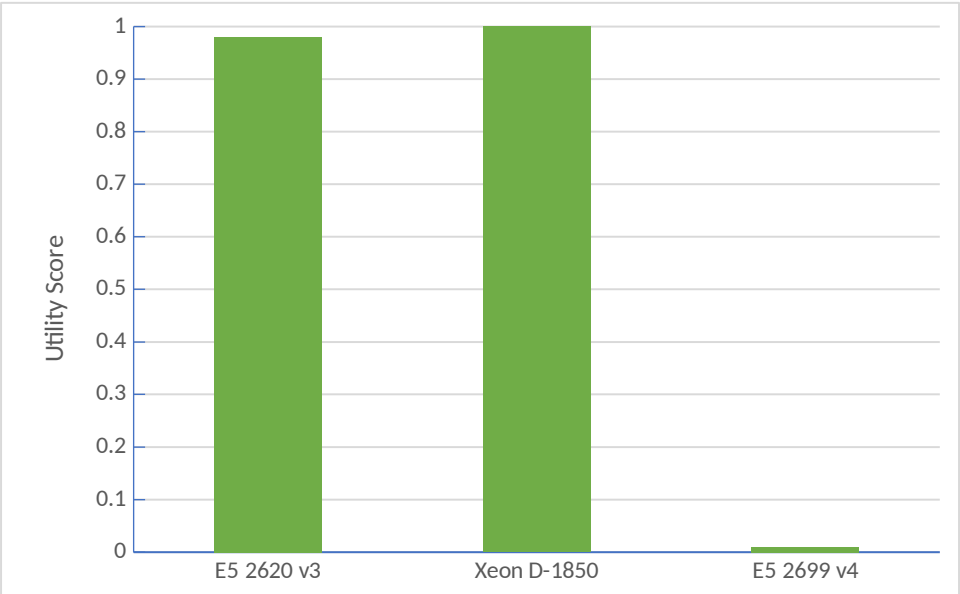


Figure 5: Economic utility score across the heterogeneous compute resources.

The distribution utility was then calculated as shown in figure 6. The Xeon D node was identified as having the highest distribution utility.

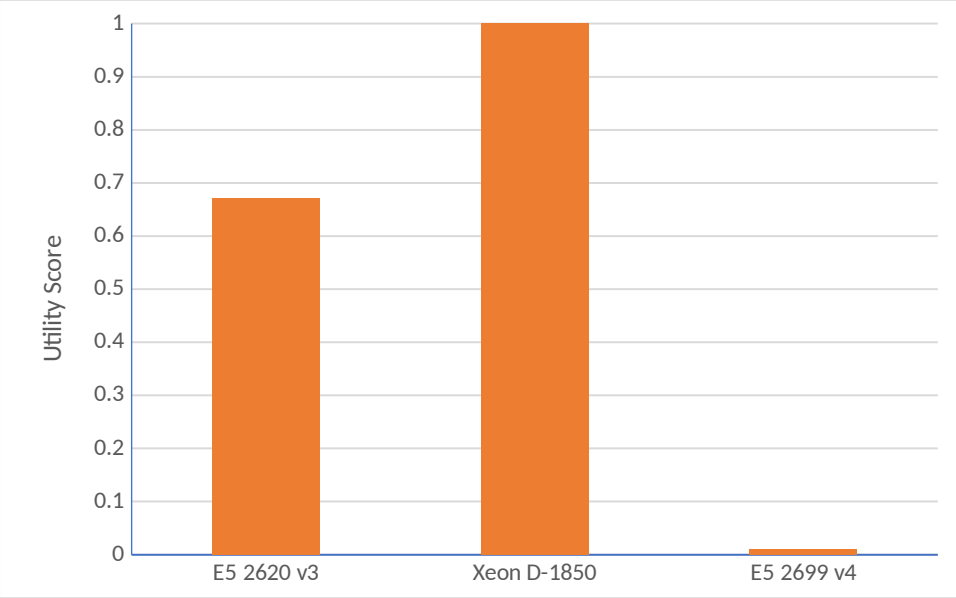


Figure 6: Distribution utility score across the heterogeneous compute resources.

Finally placement decisions based on multiple-utility provider objectives were examined. In real world deployments service providers often need to balance multi competing objectives in order to meet the SLA/KPIs for a given service deployment against the best use of their data centre infrastructure resources and provide the service at the lowest cost in order maximise profitability. A weighted approach was devise for examining the selected deployment options across the three compute nodes. Figure 7 shows three examples scenario when examining cost (economic) vs performance trade-offs. When the economic and performance utility are given equal weighting the E5 2620 compute node is the most suitable for deploying the service. When performance is given the highest weighting the E5 2699 is selected as the most suitable node. When economic utility is given the highest weighting the Xeon D had the highest utility while the E5 2620 had only a slightly lower utility.

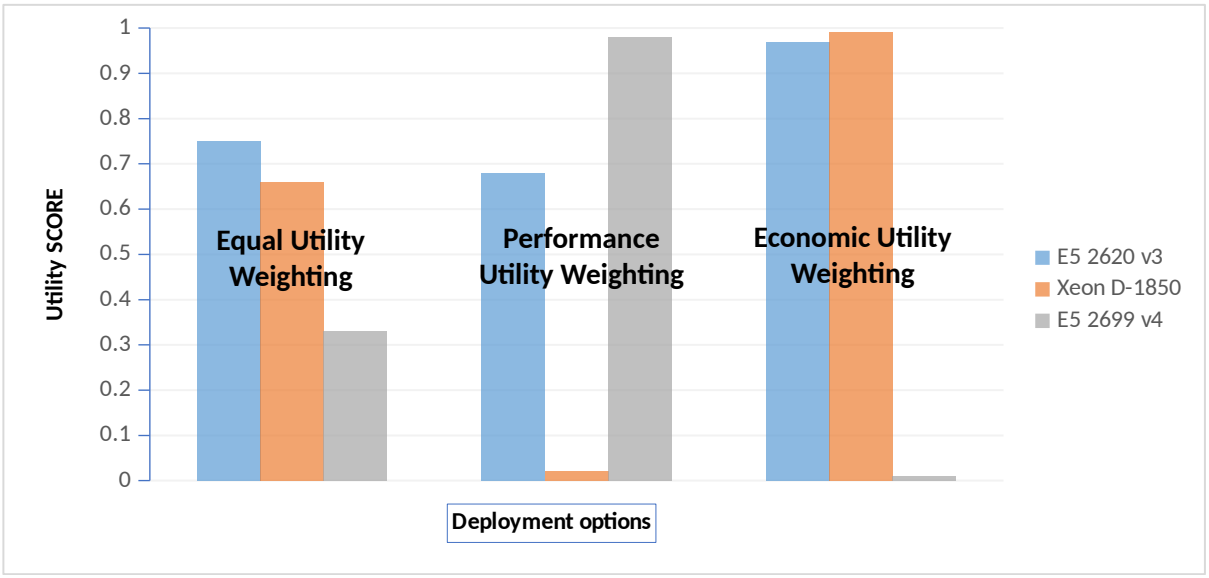


Figure 7: Utility Driven Placement across Heterogeneous Compute Resources using provider weightings.

2.9 Discussion and Conclusions

The application of utility functions to study a multi-optimisation deployment problem has been applied in the context of initial placements for on-boarding a video transmuxing service in a heterogeneous compute infrastructure environment. The approach developed combines sub-utilities of multiple objectives, analysed on a common NFVI to consistently rank solutions and thus capture the benefits and shortcomings of each deployment. The definitions appear to be novel, with limited investigation available in current literature.


3 Fast Userspace Packet Processing

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

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

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

3.1 Introduction

Recent years have seen a renewed interest for software network processing. However, as will be shown in section 3.3, a standard general-purpose kernel stack is too slow for line-rate processing of multiple 10-Gbps interfaces. To address this issue, several userspace I/O frameworks have been proposed. Those allow to bypass the kernel and obtain efficiently a batch of raw packets with a single syscall, while adding other capabilities of modern Network Interface Cards (NIC), such as support for multi-queuing.

This document first reviews the technical aspects of some existing userspace I/O frameworks, such as Netmap [24], Intel's DPDK [15], OpenOnload [27], PF_RING [21], PacketShader I/O [13] and Packet_MMAP [19]. Some other works go further than a "simple" Linux module bypassing the kernel, like IX [10] and Arrakis [22]. We won't consider them as, for our purpose, they only offer fast access to raw packets, but in a more protected way than the others I/O frameworks, using virtualization techniques, and they were not fully available at the time this document was written.

To explore their performance inside a general purpose environment, we then compare the existing off-the-shelf integrations of some of these frameworks in the Click Modular Router [17]. Click allows to build routers by composing graphs of elements, each having a single simple function (e.g. decrementing a packet TTL). Packets then flow through the graph from input elements to output


elements. Click offers a nice abstraction, includes a wealth of usual network processing elements, and already has been extended for use with some of the studied I/O frameworks. Moreover, we think its abstraction may lend itself well to network stack specialization (even if it's mostly router-oriented for now).

Multiple authors proposed enhancements to the Click Modular Router. RouteBricks [11] focuses on exploiting parallelism and was one of the first to use the multi-queue support of recent NICs for that purpose. However, it only supports the in-kernel version of Click. DoubleClick [16] focuses on batching to improve overall performances of Click with PacketShader I/O [13]. SNAP [28] also proposed a general framework to build GPU-accelerated software network applications around Click. Their approach is not limited to linear paths and is complementary to the others, all providing mostly batching and multi-queuing. All these works provide useful tips and improvements for enhancing Click, and more generally building an application on top of a "raw packets" interface.

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

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

Finally, based on this analysis, and new ideas of our own such as a graph analysis to discover the path that each thread can take to minimize the use of memory and multi-queue, we propose a new userspace I/O integration in Click (including a reworked implementation for Netmap, and a novel one for Intel's DPDK). Our approach offers both simpler configuration and faster performance. The network operator using Click does not need to handle low-level hardware-related configuration anymore. Multi-queue, core affinity and batching are all handled automatically (but can still be tweaked). On the contrary to previous work which broke the compatibility by requiring a special handling of batches, our system is retro-compatible with existing Click elements. The Click developer is only required to add code where batching would improve performance, but it's never mandatory.


This new implementation is available at [9] and will conclude our contribution with the fastest version of Click we could achieve on our system, more flexible and easy-to-use with regard to modern techniques such as NUMA-assignement, multi-queuing, core and interrupt affinity or configuration of the underlying framework.

Section 3.2 reviews a set of existing userspace I/O frameworks. Section 3.3 then evaluates their forwarding performance. Section 3.4 discusses how some of these frameworks were integrated into the Click modular router. Section 3.5 analyzes those integrations, and various improvements to Click, giving insights for the design of fast userspace packet processors. We then propose FastClick, based on those insights. Finally, section 3.6 evaluates the performance of our implementation and section 3.7 concludes this section about FastClick.

3.2 I/O Frameworks

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

3.2.1 Features

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

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

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


Kernel Bypass. Modern operating system kernels provide a wide range of networking functionalities (routing, filtering, flow reconstruction, etc.). This generality does, however, comes at a performance cost which prevents to sustain line-rate speed in high-speed networking scenarios (either multiple 10-Gbps NICs, or rates over 10 Gbps). To boost performance, some frameworks bypass the kernel altogether, and deliver raw packet buffers straight into userspace. The main drawback of this approach is that this kernel bypass also bypasses the native networking stack; the main advantage is that the needed userspace network stack can be optimized for the specific scenario [18].

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

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

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

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

3.2.2 Frameworks

Packet_mmap [19] is a feature added to the standard UNIX sockets in the Linux kernel[2], 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

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

kernel stack and need an in-kernel copy between the DMA buffer and the sk_buff, only the kernel to user-space is avoided and vice versa.

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

Table 2: I/O frameworks features summary.

PacketShader [13] is a software router using the GPU as an accelerator. For the need of their work, the authors implemented PacketShader I/O, a modification of the Intel IXGBE driver and some libraries to yield higher throughput. It uses pre-allocated buffers, and supports batching of RX/TX packets. While the kernel is bypassed, packets are nevertheless copied into a contiguous memory region in userspace, for easier and faster GPU operations.

Netmap [24] provides zero-copy, kernel bypass, batched I/O and support for multi-queuing. However, the buffer pool allocated to an application is not dynamic, which could prevent true zero-copy in some scenarios where the application must buffer a lot of packets. Recently, support for pipes between applications has also been added. Netmap supports multiple device families (IGB, IXGBE, r8169, forcedeth, e1000 and e1000e) but can emulate its API over any driver at the price of reduced performance.

PF_RING ZC (ZeroCopy) [21] is the combination of ntop's PF_RING and ntop's DNA/LibZero. It is much like Netmap [20], with both frameworks evolving in the same way, adding support for virtualization and inter-process communication. PF_RING ZC has also backward compatibility for non-modified drivers, but provides modified drivers for a few devices. The user can choose to detach an interface from the normal kernel stack or not. As opposed to Netmap, PF_RING ZC supports huge pages and per-NUMA node buffer regions, allowing to use buffers allocated in the same NUMA node than the NIC.


A major difference is that PF_RING ZC is not free while Netmap is under a BSD-style license. The library allows 5 minutes of free use for testing purpose, allowing us to do the throughput comparison of section 3.4 but no further testing. Anyway, the results of this paper should be applicable to PF_RING DNA/ZC.

DPDK. The Intel Data Plane Development Kit [15] is somehow comparable to Netmap and PF_RING ZC, but provides more user-level functionalities such as a multi-core framework with enhanced NUMA-awareness, and libraries for packet manipulation across cores. It also provides two execution model: a pipeline model where typically one core takes the packets from a device and give them to another core for processing, and a run-to-completion model where packets are distributed among cores using RSS, and processed on the core which also transmits them.

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

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

OpenOnload [27] is comparable to DPDK but made by SolarFlare, only for their products. It also includes a user-level network stack, allowing to quickly accelerate existing applications.

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

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

3.3 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 [14]. The motherboard is an Asus P9X79-E WS [7]with 44 GB of RAM at 1.6 GHz in Quad-Channel mode. We use 2 Intel (dual port) X520 DA cards for our performances tests. Previous experiments showed that those Intel 82599-based cards cannot receive small packets at line-rate, even with the tools from framework's author [13][24]. Experiments done for figure 8 lead to the same conclusion, our system seems to be capped to 33 Gbps with 64-byte packets.

To be sure that differences in performances is due to changes in the tested platform, a Tilera TileENCORE Gx36 card fully able to reach line-rate in both receive and transmit side was used. We used a little software of our own available at [9] to generate packets on the Tilera at line-rate towards the computer running the tested framework connected with 4 SFP+ DirectAttach cables, and count the number of packets received back. All throughput measurements later in this paper indicate the amount of data received back in the Tilera, 40 Gbps meaning that no loss occurred, and a value approaching 0 that almost all packets were lost. We start counting after 3 seconds to let the server reach stable state and compute the speed for 10 seconds. The packets generated have different source and destination IP addresses, to enable the use of RSS when applicable.

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

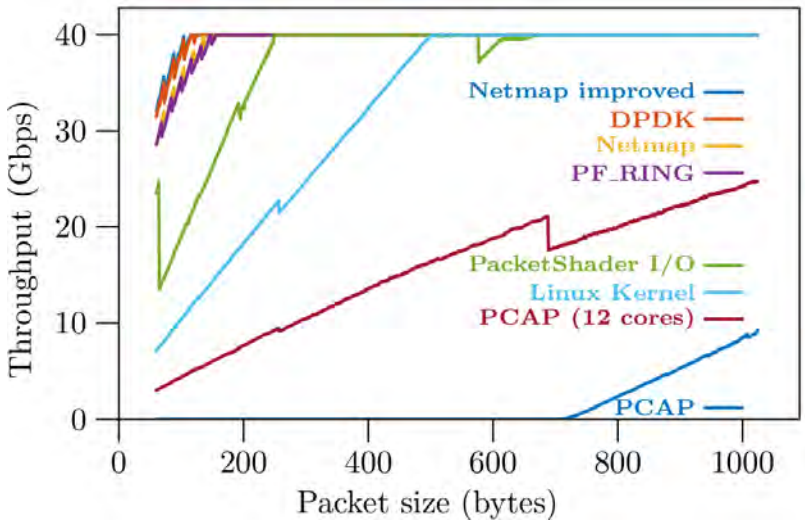


Figure 8: Forwarding throughput for some I/O frameworks using 4 cores and no multi-queuing.

We realized that our NICs have a feature whereby the status of a transmit packet ring can be mirrored in memory at very low performance cost. We modified Netmap to exploit this feature and also limited the release rate of packets consumed by the NIC to only recover buffer for sent packets once every interrupt, which released the PCIe bus of useless informations and brought Netmap performances above DPDK as we can see in figure 8. For 64-


byte packets, theses improvements boost the throughput by 14%, 1.5% over DPDK. However, except for the line labeled “Netmap Improved” in figure 8, only the final evaluation in section 3.6 use this improved Netmap version.

As expected (because they share many similarities), PF_RING has performance very similar to (standard) Netmap as shown in figure 8.

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

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

We also made a little Linux module available at [9] which transmits all received packets on an interface back to the same interface directly within the interrupt context. It should show the better performances that the kernel is able to provide in the same 4 cores and no multi-queue conditions. The relative slowness of the module regarding the other frameworks can be explained by the fact the receive path involves the building of the heavy skbuff and the whole path involves multiple locking, even if in our case no device is shared between multiple cores.

PCAP shows very poor performances because it does not bypass the kernel like the other frameworks and, in addition to the processing explained for the Linux module, the packet needs to go through through the kernel forward information base (FIB) to find its path to the PCAP application. But the bigger problem is that it relies on the kernel to get packets, and each IRQ cause too much processing by the kernel, which is overwhelming a single core and does not let enough time for any thread actually consuming the packets to run, even with NAPI[3] enabled.

This is known as a receive live-lock and [25] shows that beyond a certain packet rate the performance drops. The solution is either to use a polling system like in FreeBSD [23] or DPDK, or reduce the “per-packet” cost in the IRQ routines like in Netmap where it does very little processing (e.g. flags and ring buffer counter updates), or to distribute the load using techniques like multi-queue and RSS to ensure that each core receives fewer packets than the

3 NAPI, which stands for “New API” is an interface to network device drivers designed for high-speed networking via interrupt mitigation and packet throttling [26].


critical live lock threshold. Our kernel module is not subject to the receive live lock problem because the forwarding of the packet is handled in the IRQ routine which does not interrupt itself.

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

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

3.4 I/O integrations in Click

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

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

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

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

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


- Vanilla Click + Linux Socket: this is our off the shelf baseline configuration. The Linux socket does not expose batching, so I/O batching is not available to Click.
- Vanilla Click + PCAP: while PCAP eliminates the kernel to userspace copy by using packet_mmap, Click still copies the packets from the PCAP userspace buffer into its own buffers. However, PCAP uses I/O batching internally, only some of the library calls from Click produce a syscall.
- Vanilla Click + Netmap: as netmap exposes hardware multi-queues, these can appear as distinct NICs to Click. Therefore multi-queue configuration can be achieved by using one Click element per hardware queue. Netmap exposes I/O batching, so Click uses it.
- DoubleClick [16]: integrates PacketShader I/O into a modified Click. The main modification of Click is the introduction of compute batching, where batches of packets (instead of individual packets) are processed by an element of a task, before passing the whole batch to the next element. PacketShader I/O exposes I/O batching and supports multi-queuing.
- Kernel Click: To demonstrate the case for userspace network processing, we also run the kernel-mode Click. We only modified it to support the reception of interrupts to multiple cores. Interrupt processing (creating a skbuff for each incoming packets) is very costly and using multiple hardware queues pinned to different cores allows to spread the work. Our modification has been merged in the mainline Click [8].

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

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

Table 3: Click integrations of I/O frameworks.


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

We ran tests for pure packet forwarding, similar to those in section 3.2, but through Click. Each packet is taken from the input and transmitted to the output of the same device. The configuration is always a simple FromDevice pushing packets to a ToDevice. These two elements must be connected by a queue, except in DoubleClick where the queue is omitted (and thus the FromDevice and ToDevice elements run in the same task) because PacketShader I/O does not support the select operation. As a result, the ToDevice in DoubleClick cannot easily check availability of space in the output ring buffer, while the FromDevice continuously polls the input ring buffer for packets. As soon as the FromDevice gets packets, these are thus completely processed in a single task.

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

In all scenarios, corresponding FromDevice and ToDevice are pinned to the same core. The results are shown in figure 9. For this simple forwarding case, compute batching does not help much as the Click path consists of a very small pipeline and the Netmap integration already takes advantage of I/O Batching. Therefore Netmap closely follows DoubleClick.

The in-kernel Click, the integration of Click with PCAP and the one using Linux Socket all showed the same receive live lock behavior than explained in section 3.3. The same configurations where interrupt requests (IRQ) are dispatched to 8 logical cores and keep 4 logical cores to run Click lead to the best performances for those 3 frameworks.

The in-kernel Click is running using kernel threads and is therefore likely to receive live lock for the same reason than the PCAP configuration in section 3.3. The interrupts do less processing than for sockets because they do not pass through the forward information base (FIB) of the kernel but still create heavy skbuff for each packet and call the “packet handler” function of Click with a higher priority than Click’s thread themselves, causing all packet to be dropped in front of Click’s queue while nearly never servicing the queue consumer.

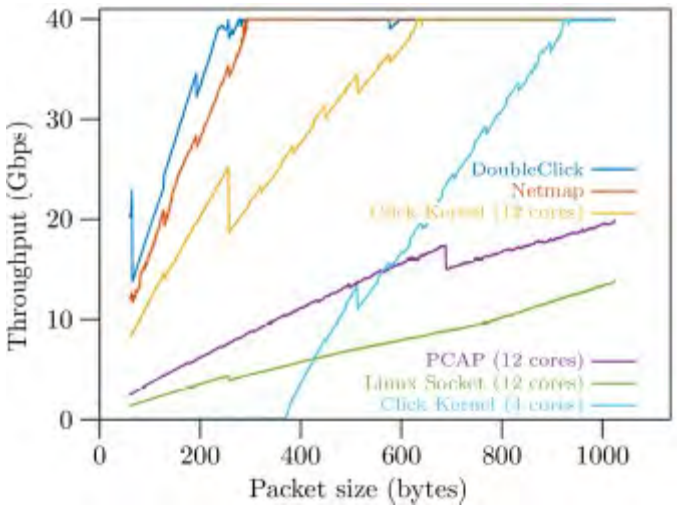


Figure 9: Forwarding throughput for some Click I/O implementations with multiples I/O systems using 4 cores except for integration heavily subject to receive live lock.

We also tested a router configuration similar to the standard router from the original Click paper, changing the ARP encapsulation element into a static encapsulation one. Each interface represents a different subnetwork, and traffic is generated so that each interface receives packets destined equally to the 3 others interfaces, to ensure we can reach line-rate on output links. As the routing may take advantage of flows of packets, having routing destination identical for multiple packets, our generator produces flows, that is packets bearing an identical destination, of 1 to 128 packets. The probability of a flow size is such that small flows are much more likely than large flows (fig. 10).

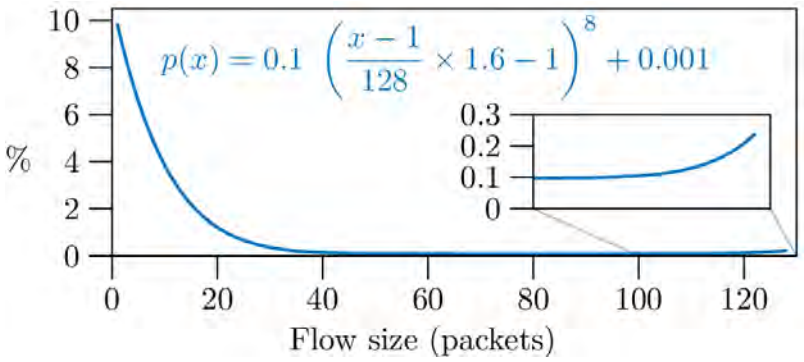


Figure 10: Probability of having flow of 1 to 128 packets for the router packet generator.

Results are shown in figure 11. We omit the PCAP and socket modes as their performance is very low in the forwarding test. Additionally, we show the Linux kernel routing functionality as a reference point.

DoubleClick is faster than the Netmap integration in Click, owing to its compute batching mode and its single task model. They are both faster than the Linux native router as it does much more processing to build the skbuffs and go

through the FIB than Click which does only the minimal amount of work for routing. The Kernel-Click is still subject to receive live lock and is slower than the native kernel router when routing is done on only 4 cores. Even when using 12 cores, Kernel-Click is slower than DoubleClick and the Netmap integration.

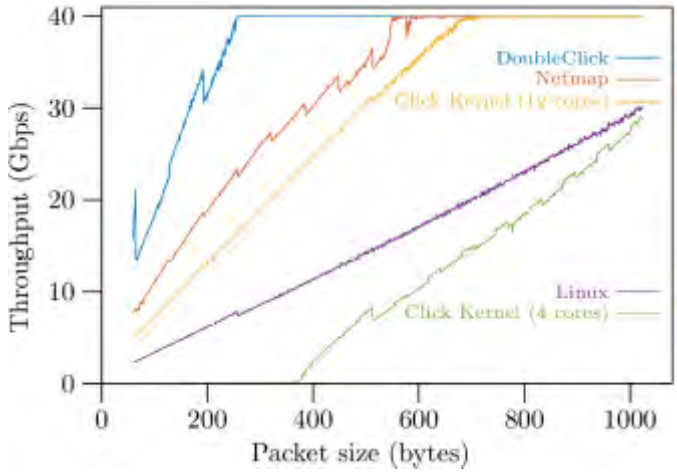


Figure 11: Throughput in router configuration using 4 cores except for in-kernel Click.

3.5 Analysis towards FastClick

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

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

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

3.5.1 I/O Batching

Both DPDK and Netmap can receive and send batches of packets, without having to do a system call after each packet read or written to the device. Figure 12 assess the impact of I/O batching using Click in a modified version to force the synchronization call after multiple batch size in both input and output.

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

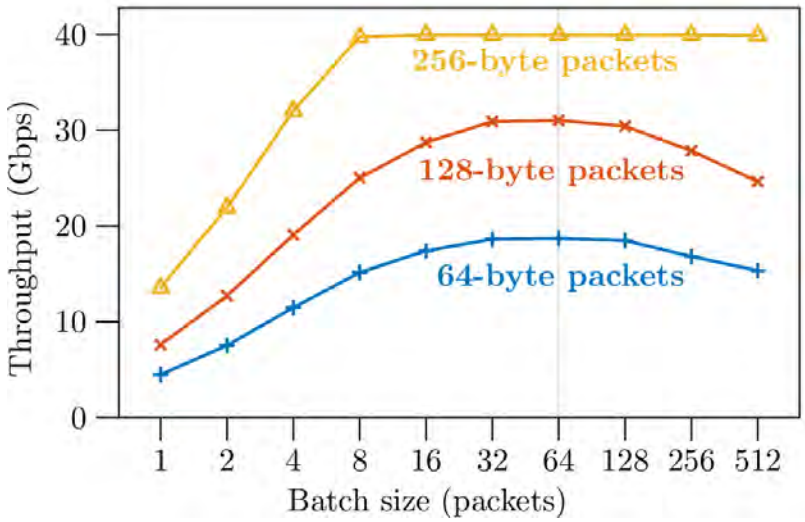


Figure 12: I/O Batching – Vanilla Click using Netmap with 4 cores using the forwarding test case (queue size = 512). A synchronization is forced after each "batch size".

3.5.2 Ring Size

The burst limit is there for insuring that synchronization is not done after too few packets. As such it should not be related to the ring size. To convince ourselves, we ran the same test using FastClick with multiple ring sizes and found that the better burst choice is more or less independent to the ring size as shown in figure 13.

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

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

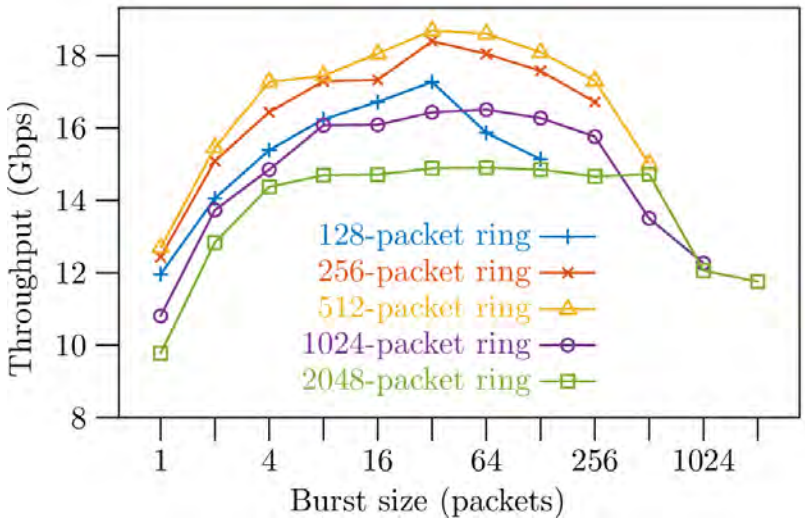


Figure 13: Influence of ring size - FastClick using Netmap with 4 cores using the forwarding test case and packets of 64 bytes.

3.5.3 Execution Model

In the standard Click, all packets are taken from an input device and stored in a FIFO queue. This is called a “push” path as packets are created in the “FromDevice” elements and pushed until they reach a queue. When an output “ToDevice” element is ready (and has space for packets in the output packet ring it feeds), it traverses the pipeline backwards asking each upstream element for packets. This is called a “pull” path as the ToDevice element pulls packets from upstream elements. Packets are taken from the queue, traverse the elements between the queue and the ToDevice and are then sent for transmission as shown in figure 14 (a).

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

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

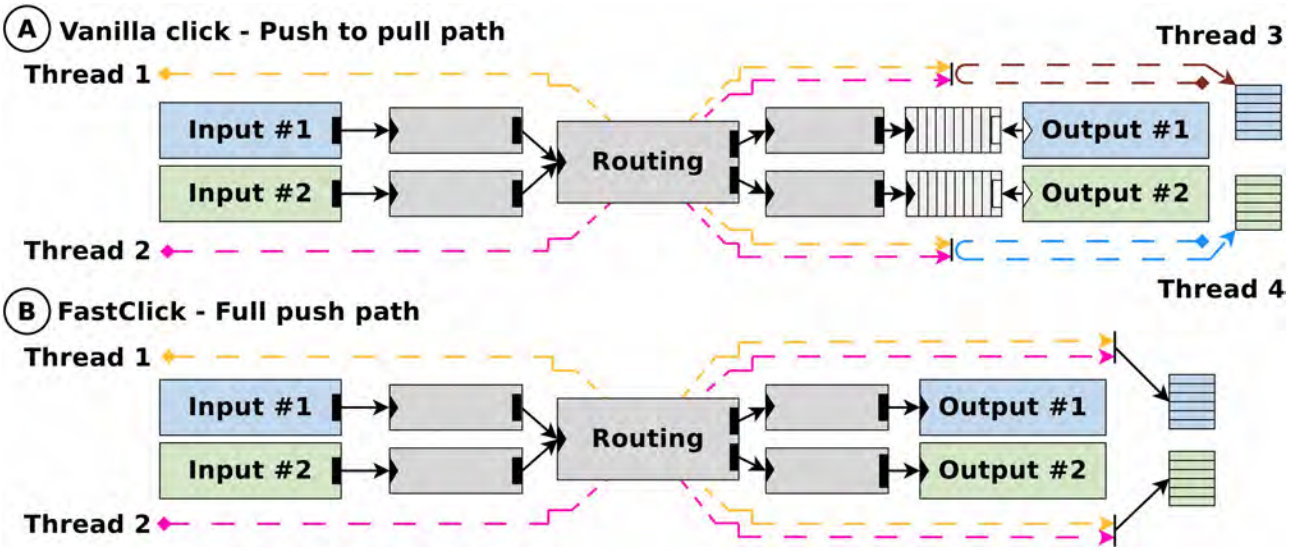


Figure 14: Push to Pull and Full Push path in Click.

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

NICs now possess receive and transmit rings with enough space to accommodate up to 4096 packets for the Intel 82599-based cards (not without some cost as seen in section 3.5.2), so these are sufficient to absorb transient traffic and processing variations, substituting advantageously for the Click queues.

Packets are taken from the NIC in the FromDevice Element and packets are pushed one by one into the Click pipe, even if I/O batching is supported. It does so until it reaches a ToDevice Element and adds it in the transmit buffer as shown in figure 14 (b). If there is no empty space in the transmit ring, the thread will either block or discard the packets.

This method is introducing 3 problems.

- All threads can end up in the same output elements. So locking must be used in the output element before adding a packet to the output ring.
- Depending on packet availability at the receive side, packets are added to the output rings. But the output ring must be synchronized sometime, to tell the NIC that it can send some packets. Syncing too often cancels the gain of batching, but syncing too sporadically introduces latency.

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

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

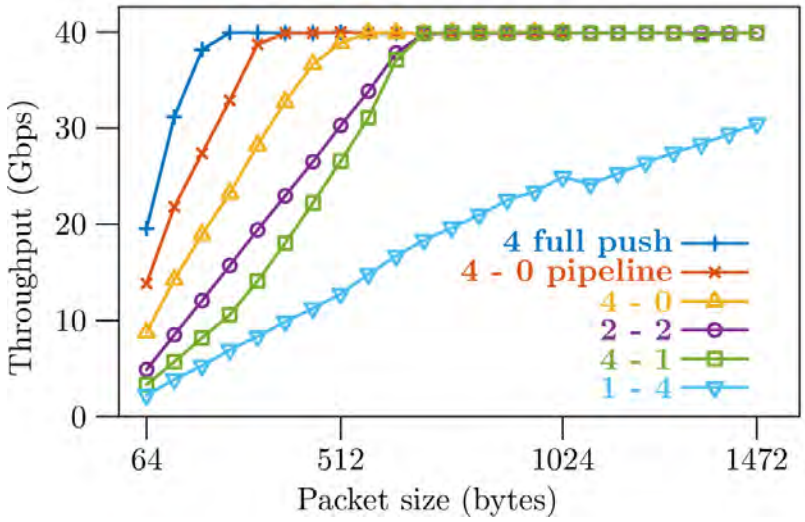


Figure 15: Comparison between some execution models. Netmap implementation with 4 or 5 cores using the router test case.

Figure 15 shows a performance comparison using the Netmap implementation with I/O batching, and a varying number of cores running FromDevice and ToDevice (the label $i-j$ represents i cores running the 4 FromDevice and j cores running the 4 ToDevice). The full push, where we have a FromDevice and all the ToDevice in a single thread on each core, performs best. The second best configuration corresponds to also a FromDevice and all the ToDevice running on the same core, but this time as independent Click tasks with a Click queue in between (label 4-0). Even when using 5 cores, having one core taking care of the input or the output expectedly results in a CPU constrained configuration.

While full-push mode seems best for our usage, one could need the “pipeline” mode anyway, having one thread doing only one part of the job, by using Queue element. But even in this case the full push execution model prove to be faster as shown in figure 15 by the line labeled “4 - 0 pipeline”. Using our new Pipeliner element which can be inserted between any two Click elements there is no more “pull” path in the Click sense. Packets are enqueued into a per-thread queue inside the pipeliner element and it is the thread assigned to empty all the internal queues of the pipeliner element which drives the packet through the rest of the pipeline.


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

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

Click allows to clone packets by keeping a reference to another packet as the one containing the real data, using a reference counter to know when a data packet can be freed. In the vanilla Click, the packet can be cloned and freed by multiple cores, therefore an atomic operation has to be used to increment and decrement the reference counter. In full push mode, we know that it is always the same core which will handle the packets, therefore we can use normal increment and decrement operations instead of the atomic ones. That modification showed a 3% improvement with the forwarding test case and a 1% improvement with the router test case. FastClick automatically detect a full push configuration using the technique described in the section 3.5.6.

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

3.5.4 Zero Copy

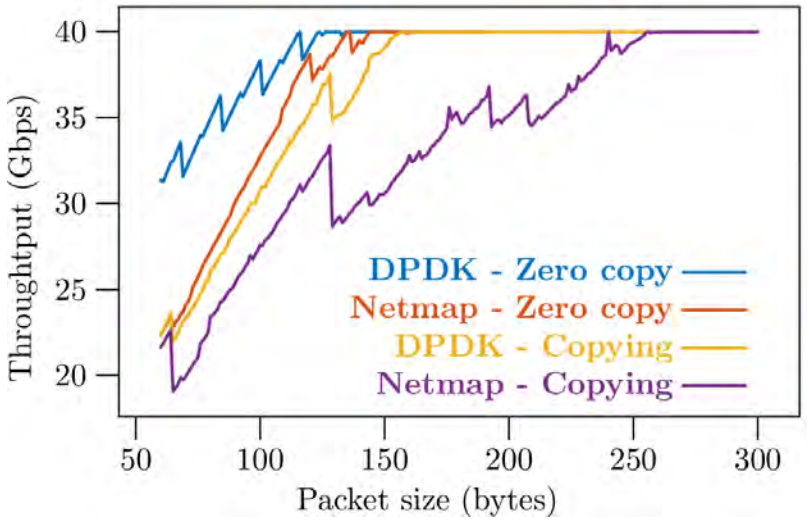


Figure 16: Zero copy influence - DPDK and Netmap implementations with 2 cores using the forwarding test case.

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

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

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

We implemented an ioctl using the technique introduced in SNAP [28] to allocate a number of free buffers from the Netmap buffer space. A pool of Netmap buffers is initialized, substituted for the Click buffer pool. When a packet is received, the Netmap buffer from the receive ring is swapped for one of the buffers from the Click buffer pool.

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

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

We used the forwarding test case as our first experiment, with only two cores to serve the 4 NICs to ensure that the results are CPU-bound, and that better performance is indeed due to a better packet system. The results are shown in figure 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.

3.5.5 Multi-queuing

Multi-queuing can be exploited to avoid locking in the full-push paths. Instead of providing one ring for receive and one ring for transmit, the newer NICs provide multiple rings per side, which can be used by different threads without any locking as they are independent as shown in figure 17.

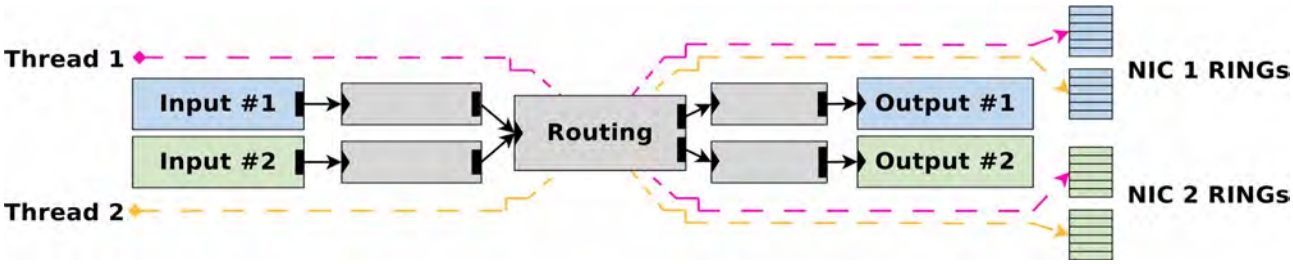


Figure 17: Full push path using multi-queue.

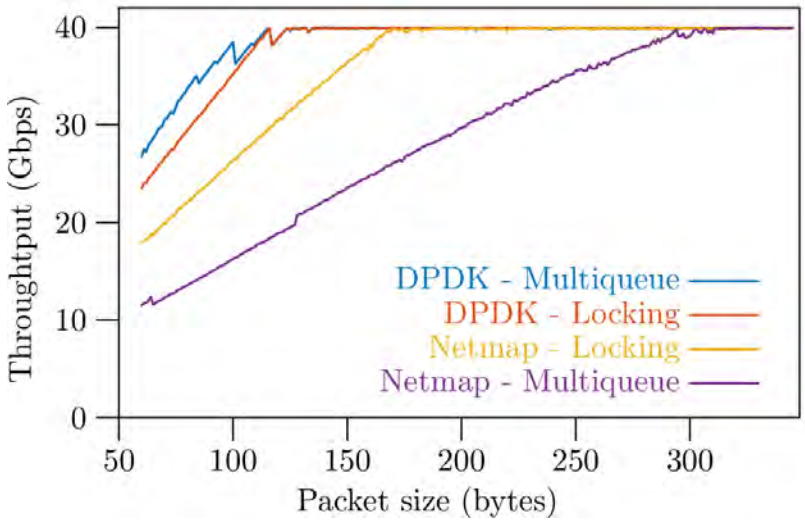


Figure 18: Full push path using multi-queue compared to locking - DPDK and Netmap implementations with 4 cores using the router test case.

We compared the full push mode using locking and using multiple hardware queues both with DPDK and Netmap, still with 4 cores. Netmap cannot have a different number of queues in RX and TX, enabling 4 TX queues forces us to look for incoming packets across the 4 RX queues. The results are shown in figure 18.

The evaluation shows that using multiple queues is slower than locking using Netmap, but provides a little improvement with DPDK. With Netmap, augmenting the number of queues produces the same results than augmenting the number of descriptors per rings, as seen in section 3.5.2. Both ends up multiplying the total number of descriptors, augmenting the size of Click's working set to the point where it starts to be bigger than our CPU's cache. As we use zero-copy, we see that the cost of reading and writing from and to Netmap's descriptors goes up with the number of queues.

3.5.6 Handling Mutable Data

In the vanilla Click one FromDevice element takes packets from one device. As show in section 3.3, handling 10 Gbps of minimal-size packets on one core is only possible with a fast framework to deliver quickly packets to userspace and a fast processor. And even in this configuration, any processing must be delegated to another core as the core running the FromDevice is nearly submerged by the reception. A solution is to use multi-queuing, not only to avoid locking like for the full push mode, but to exploit functionality such as Receive Side Scaling (RSS) which partitions the received packets among multiple queues, and therefore enables the use of multiple cores to receive packets from the same device. However, packets received in different hardware


queues may follow the same Click path. The question is thus how to duplicate the paths for each core and how to handle mutable state, that is, per-element data which can change according to the packets flowing through it, like caches, statistics, counters, etc. In figure 19, the little dots in Routing elements represent per-thread duplicable meta-data, like the cache of a last seen IP route, and the black big dot is the data which should not be duplicated because either it is too big, or it needs to be shared between all packets (like an ARP Table, a TCP flow table needing both direction of the flow, etc).

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

A first solution is to use thread-safe elements on the part of the path that multiple threads can follow as in figure 19 (a). Only mutable data is duplicated, such as the cache of recently seen routes per cores but not the other non-mutable fields of the elements such as the Forward Information Base (FIB). The advantage of this method is that in some cases memory duplication is too costly, for example in the case of a big FIB, although the corresponding data structure must become thread safe. Moreover, the operator must use a special element to either separate the path per-thread to use one output queue for each thread, or use a thread-safe queue and no multi-queue.

This is in contrast to the way SNAP and DoubleClick approach the issue: the whole path is completely duplicated, as in figure 19 (b).

A third solution would be to duplicate the element for each thread path with a shared pointer to a common un-mutable data like in figure 19 (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).

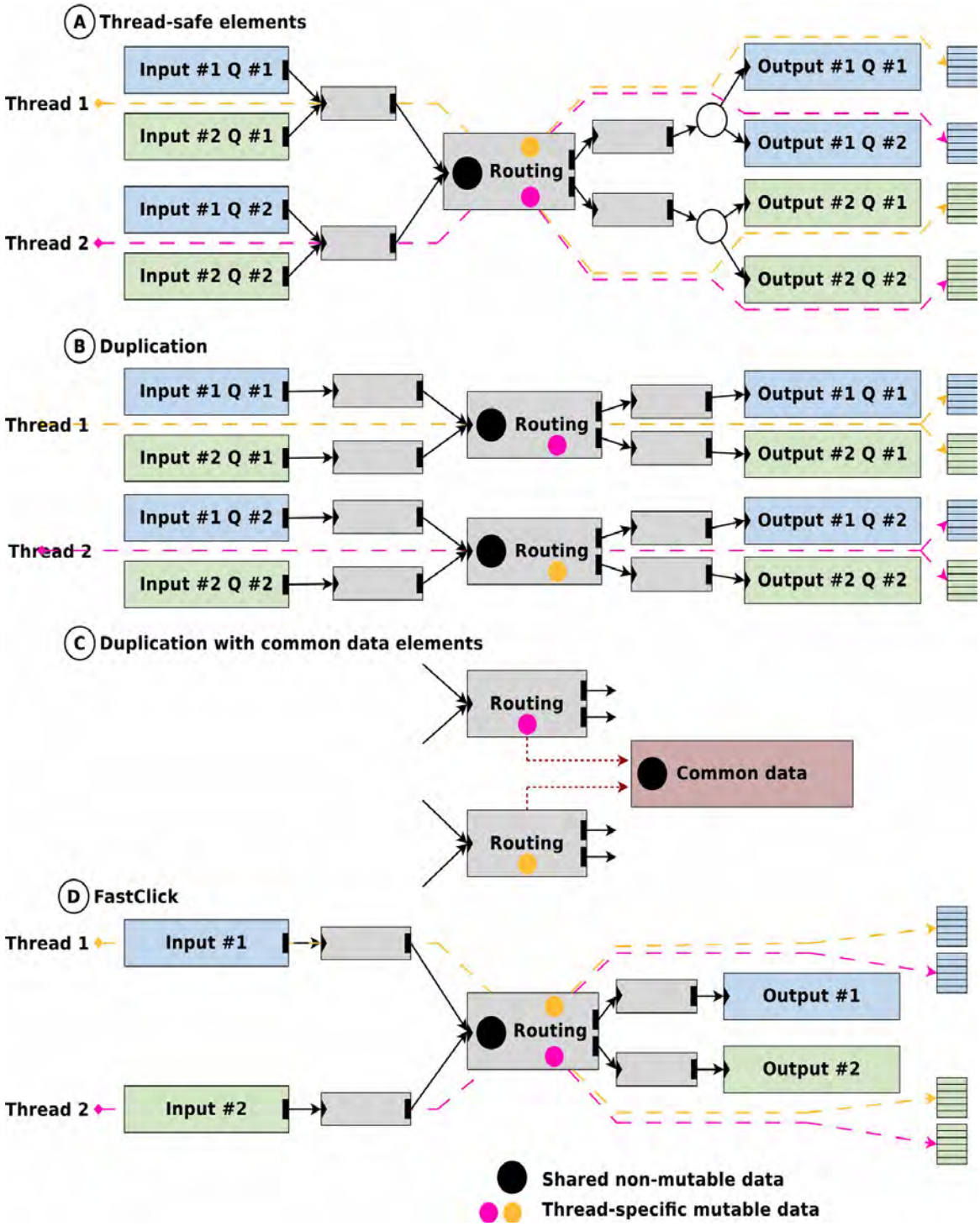


Figure 19: Three ways to handle data contention problem with multi-queue and our solution.

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


multi-queuing by assigning one hardware queue per thread as in figure 19 (d) so the operator do not need to separate path according to threads or join all threads using a queue as in figure 19 (a).

Of course, the user can still specify parameter to the FromDevice to change the number of assigned threads per input device. He can also force each thread to take care of one queue of each device to allow the same scheme than in figure 19 (a) but still with one input element and one output element per device. The difference between the two configurations depends mostly on the use case. Having each core handling one queue of each device allows more load-balancing if some NICs have less traffic than others, but if the execution paths depend strongly on the type of traffic, it could be better to have one core doing always the same kind of traffic and avoid instruction cache misses.

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

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

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

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

An example is shown in figure 20. In that example, some path contains a queue where multiple threads will push packets. As only one thread takes packet from the top right queue, the output #1 does not need to be thread-safe as only one thread will push packets into it. The output #2 will only need 3 hardware queues and not 6 (which is the number of threads used on this sample system) as the thread vector shows that only 3 threads can push packets in this element. If not enough queues are available, the thread vector allows to automatically find if the output element need to share one queue among some threads and therefore need to lock before accessing the output ring.

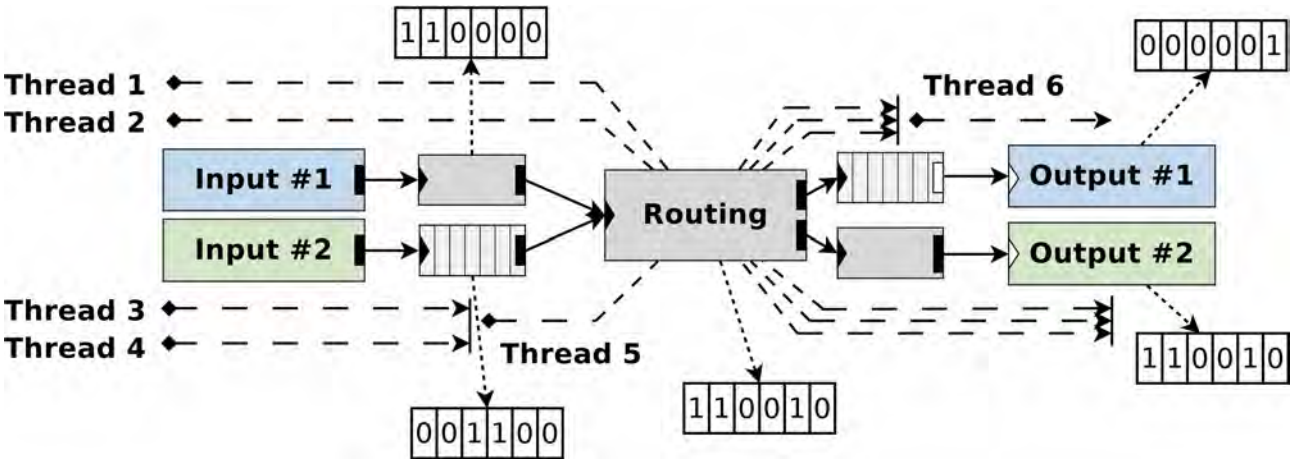


Figure 20: Thread bit vectors used to know which thread can pass through which elements.

Additionally to returning a vector, the function getThreads() can stop the router traversal if the element does not care of its input threads, such as for the queues elements. If that happens, we know that we are not in the full push mode and we'll have to use atomic operations to update the reference counter of the packets as explained in section 3.5.3.

We also provide a per_thread template using the thread vector to duplicate any structure for the thread which can traverse an element to make the implementation of thread-safe elements much more easier. Our model has the advantages of figure 19 (a) and (c) while hiding the multi-queue and thread management and the simplicity of solution (b).

3.5.7 Compute Batching

While compute batching is a well-known concept [16][28], we revisit it in the context of its association with other mechanisms. Moreover, its implementations can differ in some ways.


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

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

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

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

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

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

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

When a simple (non-batching) element has multiple output, we apply the same scheme but we have to call the start_batch() and end_batch() on all possible directly reachable BatchElement as shown in figure 21. This list is also found on configuration time.

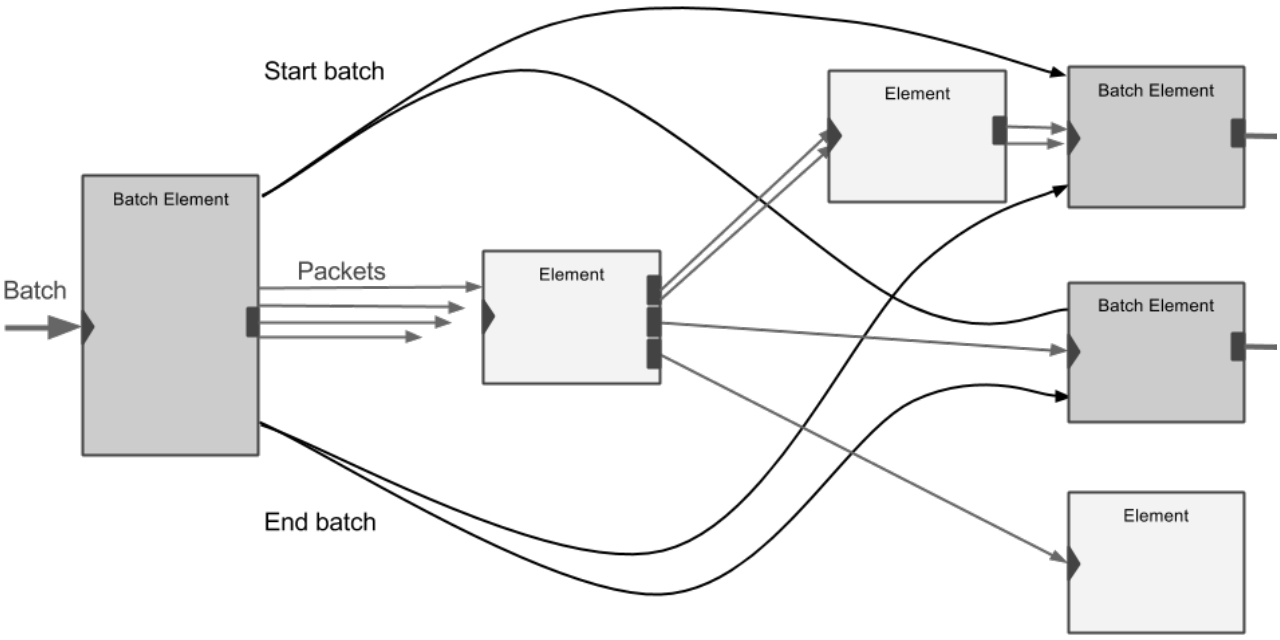


Figure 21: Un-batching and re-batching of packets when downstream elements have multiple paths.

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


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

Click keeps two pools of objects: one with Packet objects, and one with packet buffers. The previous version of Click and SNAP way of handling a freshly available Netmap packet is to take a packet from the Packet object pool and attach to the filled Netmap buffer from the receive ring. A Netmap buffer from a third pool of free Netmap buffers is then placed back in the receive ring for further reception of a new packet. SNAP does the buffer switching only if the ring is starting to fill up, maintaining multiple type of packets in the pipeline which can return to its original ring or to the third pool; introducing some management cost. We found that it was quicker to have only Netmap buffers in the buffer pool and completely get rid of Click buffer if Click is compiled with Netmap support as the allocate/free of Netmap buffer is done very often. It is very likely that if Netmap is used, packets will be either received or sent from/to a Netmap device. This is called the Netmap pool and is labeled “NPOOL” in figure 22.

Even if Netmap devices are not used, if there is not enough buffers in the pool, the pool can be expanded by calling the same ioctl than in section 3.5.4 to receive a batch of new Netmap buffers and allocate the corresponding amount of Packets. Moreover, our pool is compatible with batching and using the linked list of the batches, we can put a whole batch in the pool at once as we have only one kind of packets, this is called per-batch recycling and is labeled “RECYC” in figure 22.

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

The results of the router experiment with and without batching for both DPDK and Netmap implementations are shown in figure 22. The “BATCH” label means that the corresponding line uses batching. The “PFTCH” label means that the first cacheline of the packet is prefetched into the CPU’s cache directly when it is received in the input elements. When a packet reaches the “Classifier” element which determine the packet type by reading its Ethernet type field and dispatch packets to different Click paths, the data is already in the cache thanks to prefetching, allowing another small improvement. We omit the forwarding test case because the batching couldn’t improve the performance as it doesn’t do any processing.

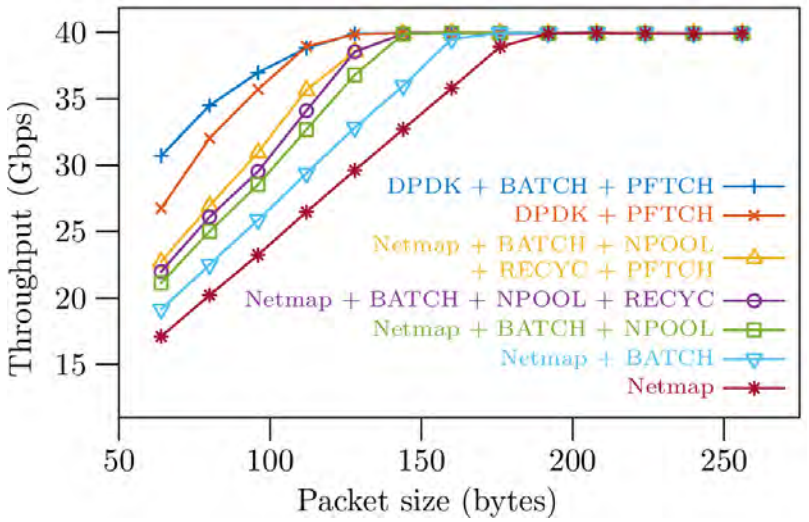


Figure 22: Batching evaluation - DPDK and Netmap implementations with 4 cores using the router test case. See section 3.5.7 for more information about acronyms in legend.

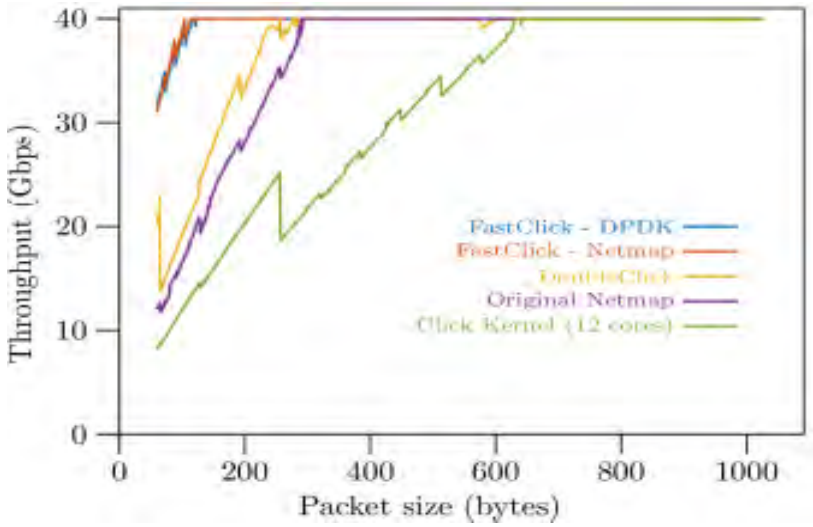


Figure 23: Comparison of forwarding throughput for FastClick and best state-of-the-art Click I/O implementations (using 4 cores except for in-kernel Click).

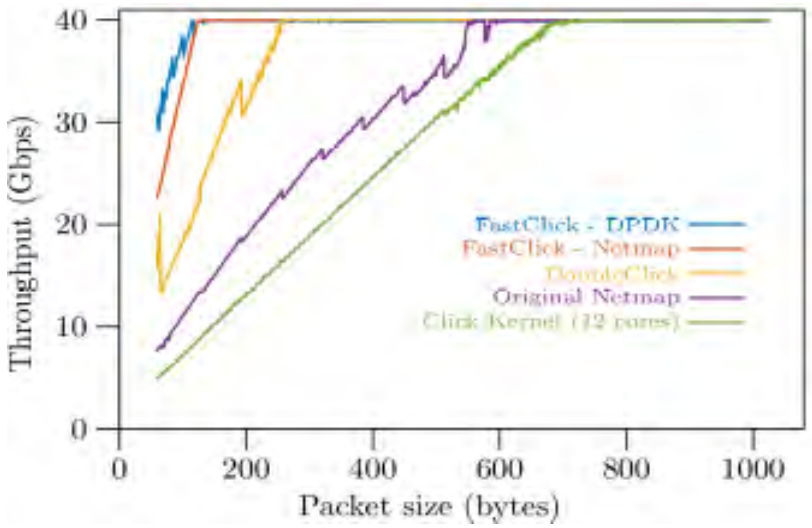


Figure 24: Comparison of router throughput for FastClick and best state-of-the-art Click I/O implementations (using 4 cores except for in-kernel Click).

3.6 FastClick evaluation

We repeated the experiments in section 3.4 with our FastClick implementation. The results of the forwarding experiments are shown in figure 23, and those of the routing experiment in figure 24.

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

An important part of the novelty is also in the configuration, which becomes much simpler (from 1500 words to 500, without any copy-paste), auto-configured according to NUMA nodes and available CPUs, and the compatibility of pipeline elements with batches. To illustrate this point, see how easily we can scale up a router application with FastClick. While Click would require a different configuration tuned for each number of allocated cores, with a lot of duplication and opportunities for errors, FastClick can use the same logical configuration file with a single description of the router, and will automatically allocate more hardware queues and execution threads as more cores become available (by specifying the `-j N_CORES` argument when launching FastClick). Figure 25 illustrates the throughput improvements as we increase the number of cores, showing the potential of easy up-scaling).

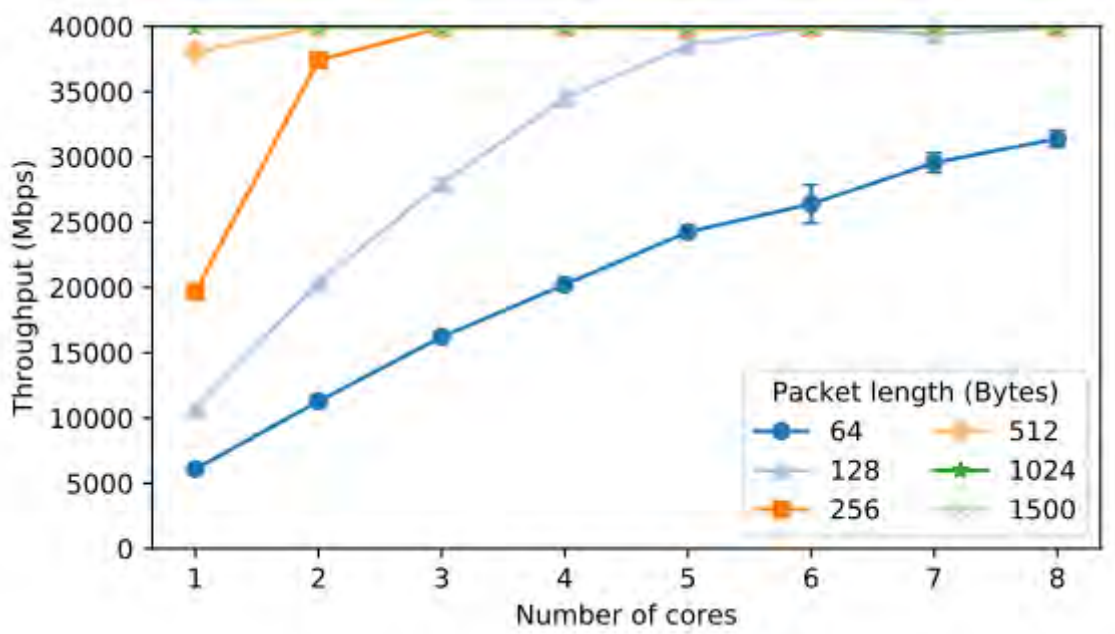


Figure 25: One argument is all it takes to scale up a FastClick router.

3.7 Conclusion

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

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

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

Beyond improved performance, FastClick also boasts improved abstractions for packet processing, as well as improved automated instantiation capabilities on modern commodity systems, which greatly simplifies the configuration of efficient Click packet processors. The automatic resource allocation strategies built into FastClick thus seem a promising first step toward the allocation of


reusable function blocks (here under the form of Click elements) pipelines onto the available resources (here, under the form of hardware machine nodes).

As many middleboxes operate on the notion of micro-flows, we have also extended the flow capabilities of FastClick to facilitate the development of such middleboxes, as explained in section 5.1.


4 SplitBox: Toward Efficient Private Network Function Virtualization

SplitBox is a scalable system for privately processing network functions in the cloud. It was developed as an example application to assess the suitability of FastClick (described in section 3) for middlebox applications. This section only briefly presents SplitBox, and focuses on the automated resource allocation using FastClick, and evaluation results. For a discussion of the cryptographic algorithms behind SplitBox, see [29].

4.1 Introduction

Network function virtualization (NFV) is increasingly being adopted by organizations worldwide, moving network functions traditionally implemented on hardware middleboxes (MBs) – *e.g.*, firewalls, NAT, intrusion detection systems – to flexible and easier to maintain software processes. Network functions can thus be executed on virtual machines (VMs), with cloud providers processing traffic destined to, or originating from, an enterprise network (the client) based on a set of policies governing the network functions. This, however, implies that confidential information as well as sensitive network policies (*e.g.*, the firewall rules) are revealed to the cloud, whereas in the traditional setting, such policies would only be known to the client's network administrators. Disclosing such policies can reveal sensitive details such as the IP addresses of hosts, the topology of the client's private network, and/or important practices [35][39].

This motivates the need to allow processing outsourced network functions without revealing the policies: we denote this problem as *Private Network Function Virtualization* (PNFV), as done in [37]. We argue that PNFV solutions should not only provide strong *security* guarantees, but also satisfy *compatibility* with existing infrastructures (*e.g.*, not requiring third parties, sending/receiving traffic, take part in new protocols) as well as *high throughput* in order to match the quality of service expected of network functions. In practice, this precludes the use of some standard cryptographic tools.

Several attempts have recently been made to support PNFV or similar functionalities [35][36][37][39], assuming the cloud to be honest-but-curious (*i.e.*, the cloud processes the network functions as instructed but may try to learn the underlying policies). However, none of these simultaneously achieve security, compatibility, and high throughput, or their coverage of network

functions is limited as they are only applicable to firewall rules that either allow or drop a packet.

Our intuition is to leverage the distributed nature of cloud VMs: rather than assuming that a single VM processes a client's network function, we distribute the functionality to several VMs residing on multiple clouds or multiple compute nodes in the same cloud. Assuming that not all VMs in the cloud are simultaneously under the control of the adversary (for instance, a *passive* attacker cannot gain access to all nodes running the distributed VMs), we are able to provide a scalable and secure solution. As discussed throughout the paper [29], achieving this solution is not straightforward and, in the process, we overcome several challenges.

We implemented and evaluated SplitBox on a firewall test case, showing that it can achieve a throughput of over 2 Gbps with 1 kB-sized packets, on average, traversing up to 60 rules.

4.2 Preliminaries

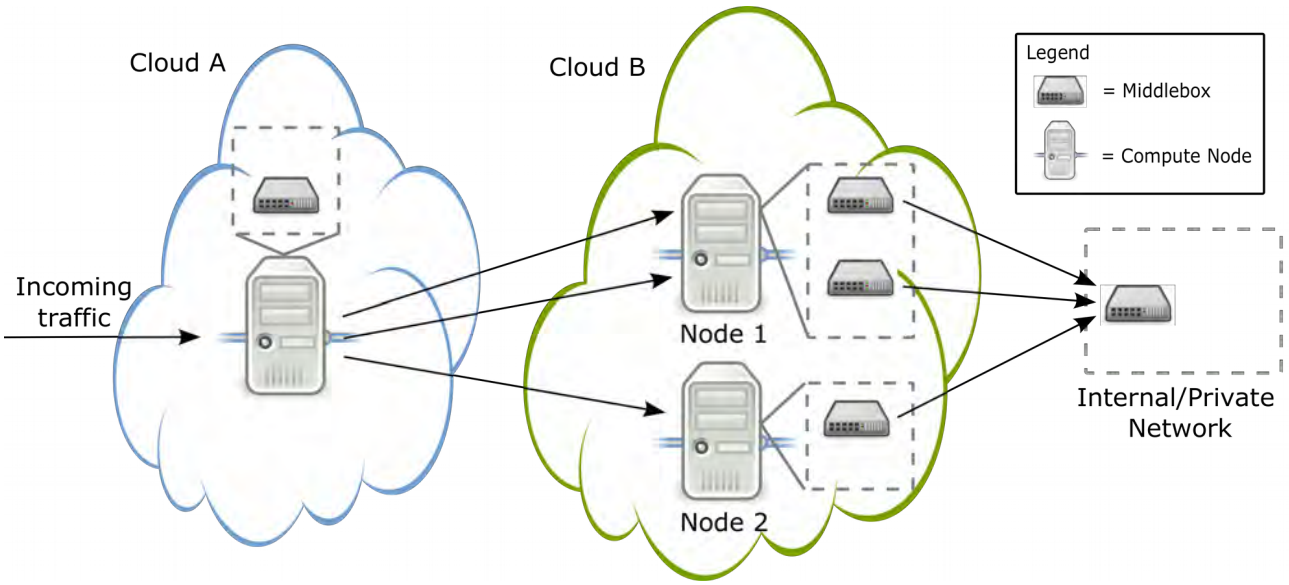


Figure 26: Our system model with Cloud A hosting MB A as a VM in one of its compute nodes. Cloud B hosts the MBs B(t) with t = 3 as VMs (not all t reside on the same compute node). The client MB C resides at the edge of the client's internal network. A and B(t) collaboratively compute network functions for the client.

System and Trust Model. Figure 26 illustrates our PNFV model, consisting of two types of cloud middleboxes (MBs): an *entry* MB $\mathscr{A}$ and $t \geq 2$ cloud MBs $\mathscr{B}(t)$, which collaboratively compute a network function on behalf of a client. The client has its own MB, denoted $\mathscr{C}$, at the edge of its internal network. $\mathscr{A}$ receives an incoming packet, does some computations on it, "splits" the result into t parts, and forwards part j to $\mathscr{B}_j \in \mathscr{B}(t)$. $\mathscr{B}_j$ performs local

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

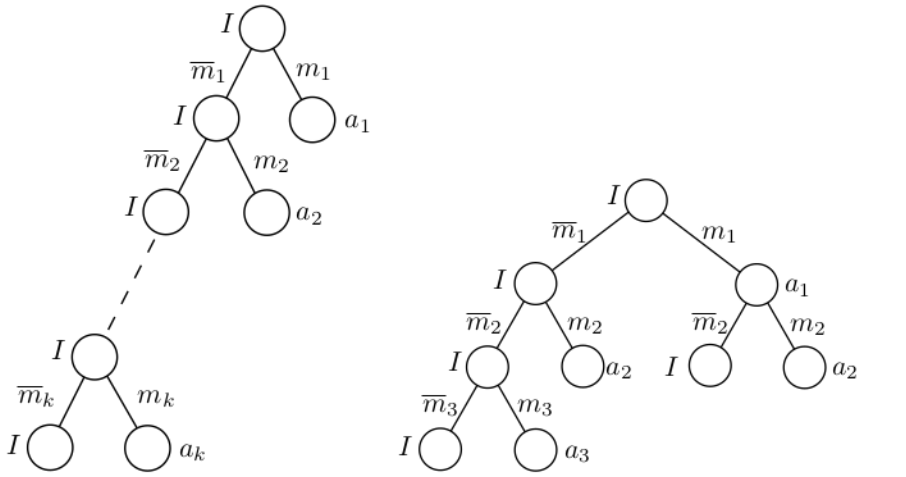
Assumptions. We assume an honest-but-curious adversary which can corrupt[4] either $\mathscr{A}$ or up to $t-1$ MBs from $\mathscr{B}(t)$, and it cannot corrupt $\mathscr{A}$ and any MB in $\mathscr{B}(t)$ simultaneously. In practice, one can assume $\mathscr{A}$ to be running on a different cloud provider than $\mathscr{B}(t)$ and that not all MBs in $\mathscr{B}(t)$ reside on the same node.

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

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

We call each pair (m,a) in ψ a *policy*. Policies serve as building blocks of a network function. The set of policies of ψ is the set of *distinct* policies (m,a) in ψ. Figure 27 (a) shows a network function with k distinct policies: whenever a match is found, the corresponding action is performed and the function terminates. The function in Figure 27 (b) has 3 distinct policies, $(m_1,a_1),(m_2,a_2)$ and (m_3,a_3), and (m_2,a_2) is repeated twice. This function does not terminate immediately after a match has been found (*e.g.*, path $m_1 m_2$). Since $a \circ I = I \circ a = a$, we can easily "plug" individual policy trees to construct more complex network functions.

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



(a) Network function with k distinct policies.

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

Figure 27: Network functions as binary trees.

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

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

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

Definitions. Throughout the rest of this section, we use the following definitions: let $z \in \{0,1,*\}^n$, the *projection* of z, denoted π_z, be a string $\in \{0,1\}^n$, such that $\pi_z(i) = 1$ if $z(i) \in \{0,1\}$ and $\pi_z(i) = 0$ if $z(i) = *$. The *masking* of a


$x \in \{0,1\}^n$ using $\pi_z \in \{0,1\}^n$, denoted $\omega(\pi_z, x)$, returns x' such that $x'(i) = x(i)$ if $\pi_z(i) = 1$ and $x'(i) = 0$ if $\pi_z(i) = 0$. $\mathbb{H} : \{0,1\}^n \rightarrow \{0,1\}^q$ denotes a cryptographic hash function; $\oplus$ denotes bitwise XOR. The Hamming weight of a string $x \in \{0,1\}^n$ is $\mathrm{wt}(x)$. Finally, $x \leftarrow_{\$} \{0,1\}^n$ means sampling a binary string of length n uniformly at random.

4.3 Introducing SplitBox

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

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

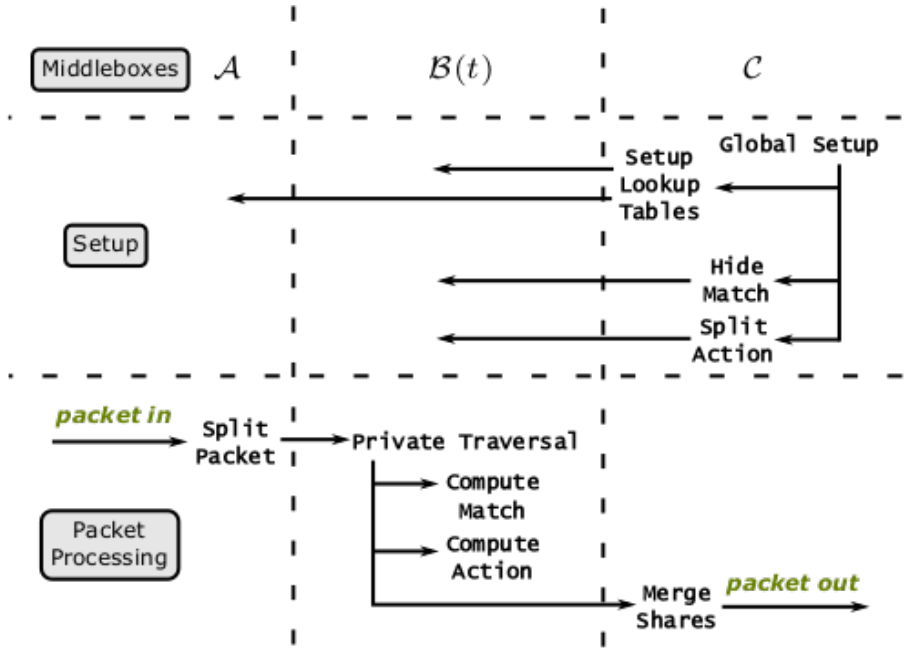


Figure 28: Breakdown of algorithms executed by each MB in SplitBox.

High-Level Overview. In a nutshell, if we assume that ψ includes a single policy (m,a), our strategy to hide m is to let $\mathscr{C}$ *blind* μ by XORing it with a random binary string s and sending the hash of the result to each MB in $\mathscr{B}(t)$; whereas, to hide a, $\mathscr{C}$ computes t *shares* of the action α using a t-out-of-t secret sharing scheme and sends share j to $\mathscr{B}_j$. In addition, $\mathscr{A}$ encrypts the contents of a packet x by XORing it with the blind s, and sends it to the MBs in $\mathscr{B}(t)$, which can then compute matches and actions on this encrypted packet. We present the details of SplitBox using a set of algorithms, grouped based on the MB executing them. Figure 28 shows a high-level overview of all the algorithms computed by each MB. We assume ψ_{priv} to be the private version of the network function ψ whose matching and action functions are replaced by unique identifiers. Once again, see [29] for details of the algorithms.

4.4 Implementation

In this section, we discuss our proof-of-concept implementation of SplitBox inside FastClick [30], an extension of the Click modular router [17] which provides fast user-space packet I/O and easy configuration via automatic handling of multi-threading and multiple hardware queues (see section 3 for details). We also use Intel DPDK [15] as the underlying packet I/O framework. We implemented three main FastClick elements: element `Entry` corresponding to MB $\mathscr{A}$, `Processor` corresponding to MBs $\mathscr{B}(t)$, and `Client` to $\mathscr{C}$. `Client` implements the `Merge Shares` algorithm, which reconstruct final packets based on information sent by MBs $\mathscr{B}(t)$. The other algorithms of $\mathscr{C}$


are executed outside the FastClick elements, and used to configure the above three elements. The hash function $\mathbb{H}$ is implemented using OpenSSL's SHA-1, aiming to achieve a compromise between security, digest length, and computation speed, as hash functions which have larger message digests will lead to overly large lookup tables. `Client` uses a circular buffer to collect packet shares until all have been received and the final packet can be reconstructed. For communication between our elements, we use UDP packets: UDP and L2 processing relies on standard Click elements such as `UDPIPEncap`. Finally, we also add a few elements to help in our delay measurements, as explained below.

To evaluate our implementation, we focus on a firewall use case, using a network function tree similar to that in Figure 27 (a). A single action is applied, either the identity action, if the packet is allowed, or marking the packet with a drop message (0^n), if it should be dropped. We use three commodity PCs for our experiments (8-core Intel Xeon E5-2630 with 2.4GHz CPU and 16 GB of RAM): one for both `Entry` and `Client`, in order to use the same clock for delay measurements, and the other two as two `Processors`. The four nodes (including the two on the same machine) are connected through Intel X520 NICs, with 10-Gbps SFP+ cables. The topology is thus very similar to the one in Figure 26, except that we only have $t=2$ in $\mathscr{B}(t)$, and that $\mathscr{A}$ and $\mathscr{C}$ share the same physical machine. Another difference is that our machines are connected directly, without intermediate routers between them. We use a trace captured at one of our campus border router (pre-loaded into memory) as input for the `Entry` element, which executes the `Split Packet` algorithm on a single core. Then, each output of `Entry` (one for $\mathscr{C}$ and one per $\mathscr{B}_j$) is encapsulated inside an UDP packet and sent to the corresponding output device, using one core per device.

On each $\mathscr{B}_j$ machine, the packets are read from the input device, decapsulated, and then passed to a `Processor` element which does the actual filtering. The resulting action packets are then re-encapsulated and sent through the NIC towards the client. This operation is done on a single core, but several cores can easily be used in parallel. With FastClick, it suffices to launch Click with more cores, and the system will automatically create the corresponding number of hardware queues on the NICs, and assign a core to each queue. On the client side, each of the three input NICs has an associated core. Incoming packets are decapsulated, and then passed to the `Client` element, which reconstructs the final packets (on its own core). Reconstructed packets which are not marked as dropped are then passed to a receiver


pipeline, which computes the entry-to-exit delay, counts packets and measures reception bandwidth. To measure delays, the packets in the in-memory list are tagged with a sequence number in the packet payload, before the transmission begins. This number allows to match the exit time-stamp with an entry time-stamp, which is kept in memory. This allows to avoid storing the time-stamp itself in the packet, which would increase the measured delay.

4.5 Performance Evaluation

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

Figure 29 illustrates the evolution of the maximum achievable bandwidth (taken as inducing less than 0.001% losses), as a function of the number of traversed rules (*i.e.* the number of match attempts per packet). Our trace packets are about 1 kB on average, so that 8 Gbps corresponds to about 1 Mpps. We observe that the bandwidth decreases significantly with more traversed rules with SplitBox (PNFV), mainly due to the hashing function, which is called on the packet header once per match attempt. Not only is this more computationally expensive than simpler comparisons, but it is also done each time on different data (as we need to first XOR packet header with match projection), taking no advantage of the cache. Fortunately, the `Processor` operation is inherently parallelizable, thus, allocating more cores speeds things up. Note that the average number of traversed rules in a real firewall is significantly lower than the total number of rules. Therefore, it is particularly important to choose the order of match attempts according to the traffic distribution, and/or to use a more complex tree structure minimizing the number of match attempts.

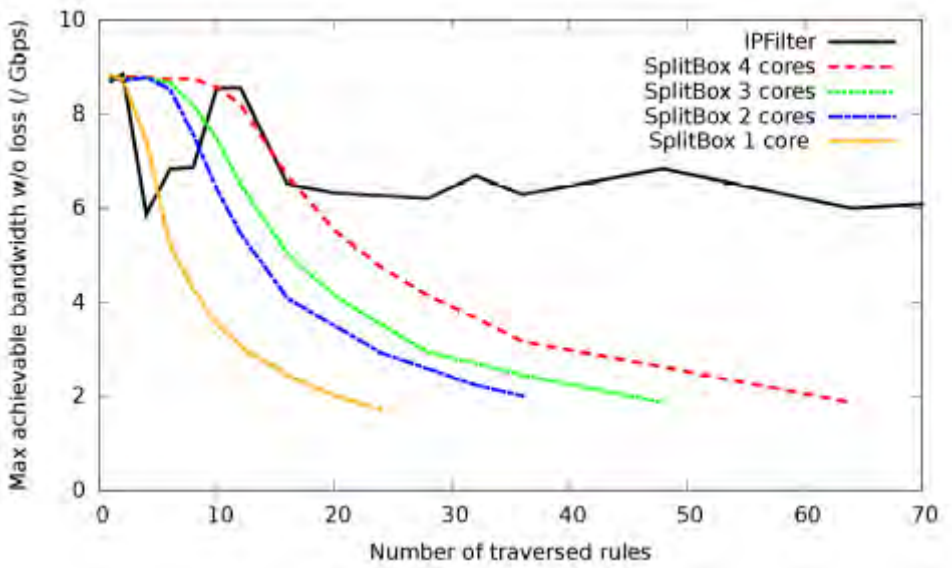


Figure 29: Achievable bandwidth drops sharply with the number of traversed rules.

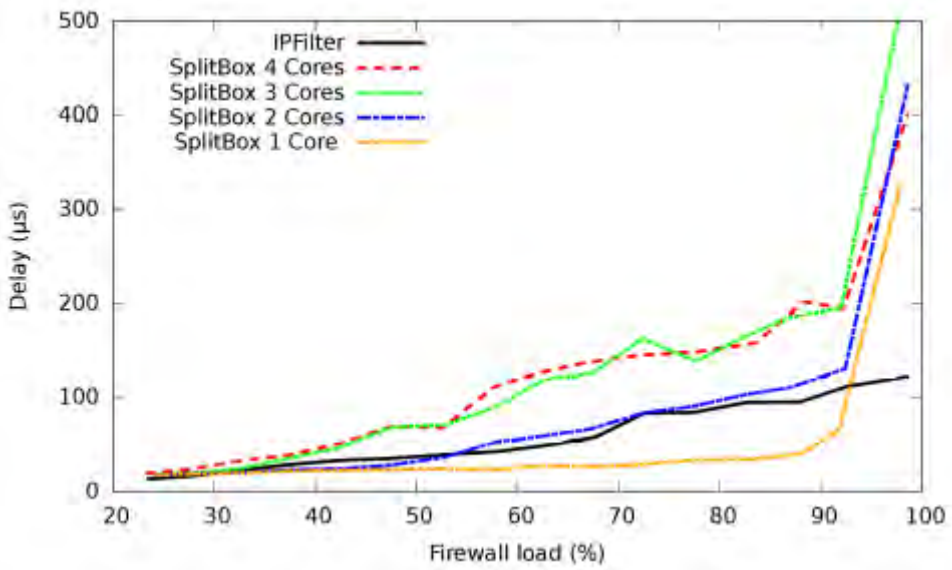


Figure 30: Delay increases with the firewall load.

Finally, in Figure 30, we plot the delays as a function of firewall load (*i.e.* current input bandwidth over maximum achievable bandwidth). Note that the delays do not follow the same dependency with regard to the number of match attempts per packet. Although these increase slightly with the number of traversed rules, they are mostly governed by queuing delays in the system (in NICs rings, or in-memory rings exchanging packets between the different processing cores). The number of blinds l seems to have little impact on the performance: with l ranging from 64 to 65,536, we observe no noticeable difference, except for additional memory consumption.


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

4.6 Conclusion & Future Work

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

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


5 Improvements to FastClick

FastClick (described in section 3) allows the easy development of fast packet processing network functions on commodity servers. Based on the Click modular router, it is a proven design, used in the industry (*e.g.* it is the basis of Cisco Meraki products), and it maps well to the Superfluidity architecture, with a direct one-to-one correspondence between reusable functional blocks (RFBs) and Click elements.

However, the reach of Click in terms in functionality is quite limited compared to VM-based network functions, which often act on higher-level flows rather than packets. It is also limited by the underlying hardware capacity of a commodity server.

In this section, we describe how we extended FastClick to address both issues, by adding support for both flow processing and hardware co-processors. As a bonus, the use of a modular architecture based on RFBs rather than stand-alone VMs allows the consolidation of some functions like classification and flow management, which leads to much better performance for service chains.

5.1 Flow processing

5.1.1 Common problems with traditional middleboxes

According to [68][69], there are roughly as many middleboxes as routers in enterprise networks. A myriad of middleboxes is also deployed in increasingly important mobile networks [70], and will be of utmost importance for 5G. These middleboxes are there for good reasons, as they provide, amongst others, network security and performance enhancements.

However, they have difficulties to cope with the growing needs for more throughput and slow down innovation, because they are often independent systems working like complete black boxes, totally unable to cooperate with other network appliances to enable the deployment of new network services. Current middlebox functionalities such as stateful firewalls, NATs, DPI, content-aware optimizers or load-balancers are often implemented in separate hardware boxes or on multiple (perhaps virtual) machines. It is common to see each packet being re-classified in each component of a service chain, even inside a single box.

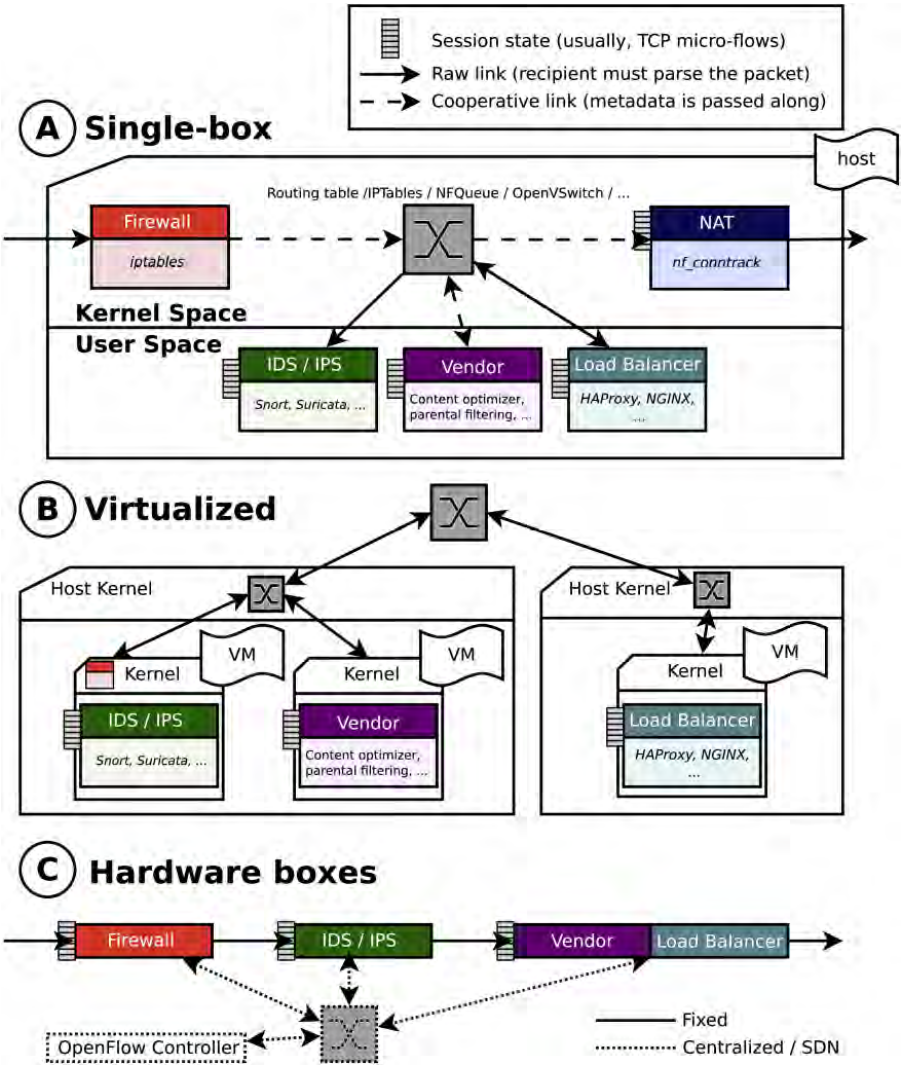


Figure 31: Different ways to build a middlebox service chain. On most links, there is no cooperation to avoid redundant operations.

Three typical implementations of a service chain are shown on figure 31. They all implement mostly the 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 [71] for IDS, NetFilter/IPTables/NFT for the firewall, HAProxy [72] or NGINX [73] for load-balancing, ... This is the setup often found in small networks.


The second one uses virtualisation and switches to dispatch packets between VMs containing mostly similar software. This setup is easier to scale and more reliable thanks to the virtualisation layer. But it also introduces penalty in performances, although recent research papers try to reduce this hit [74][75].

The third chain, mostly seen on large networks, uses different physical boxes to achieve the same results in hope of achieving better delay and throughput.

All three chains share the following issues:

- Packets are partially or completely re-classified in each middlebox component, *i.e.* packet headers are inspected to classify the packet. Classification finds the flow to which the packet belongs according to the given rules.
- A dictionary data structure is needed in all stateful middleboxes to remember per-session data.
- The chain relies on slow OS capabilities such as a generic TCP stack that may be only partially needed, and is not designed for the specific needs of middleboxes.

5.1.2 Introducing MiddleClick

We developed MiddleClick, and extension of FastClick that allows the same modularity and re-usability for middleboxes than what was available for software routers. Middlebox functionality is built by combining simple reusable function blocks (RFBs, called Elements in Click parlance), allowing the middlebox developer to concentrate on the specific functionality being developed. Moreover, MiddleClick improves performance by avoiding redundant classification work, and support offloading of (part of) the classification to some hardware device.

In MiddleClick, 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 having multiple, often identical, hash tables along the way to find the session of each packet.

By unifying the service chain both outside and inside cooperative middleboxes, we ensure that each field of the packet is looked at only once, and the **results**


of the classification and the session mapping are reused in all the following middleboxes.

The framework also provides a zero-copy **stream abstraction**, allowing to modify packets of the same session without the need for any knowledge of the different protocols. For example, when an HTTP payload is modified, the content-length must be corrected. A layered approach allows to back-propagate the effect of stream modification without knowing the implications on the bottom 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 make changes.

The system also supports a mechanism to "wait for more data" when a middlebox needs to buffer packets, unable to act while data is still missing. It supports pro-active ACKing to avoid stalling a flow while waiting for more data, and allows to handle large amount of flows using a run-to-completion-or-buffer model, which avoids costly context switches.

MiddleClick works well for software components that can be plugged in a Click-based environment in a single memory space. However, our solution can also avoid flow re-classification and ease flow management across multiple different memory spaces or boxes. Additionally, it allows to offload (part of) the classification itself to fast hardware classifiers, further improving performance.

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. For example, an OpenFlow switch can be set up to replace the VLAN ID of packets according to the type of flow.

MiddleClick was shown to introduce very little overhead over FastClick, and has very good performance in practice. For example, figure 32 shows a performance comparison between HAProxy, when used as a TCP load-balancing reverse proxy, and the same functionality implemented using MiddleClick.

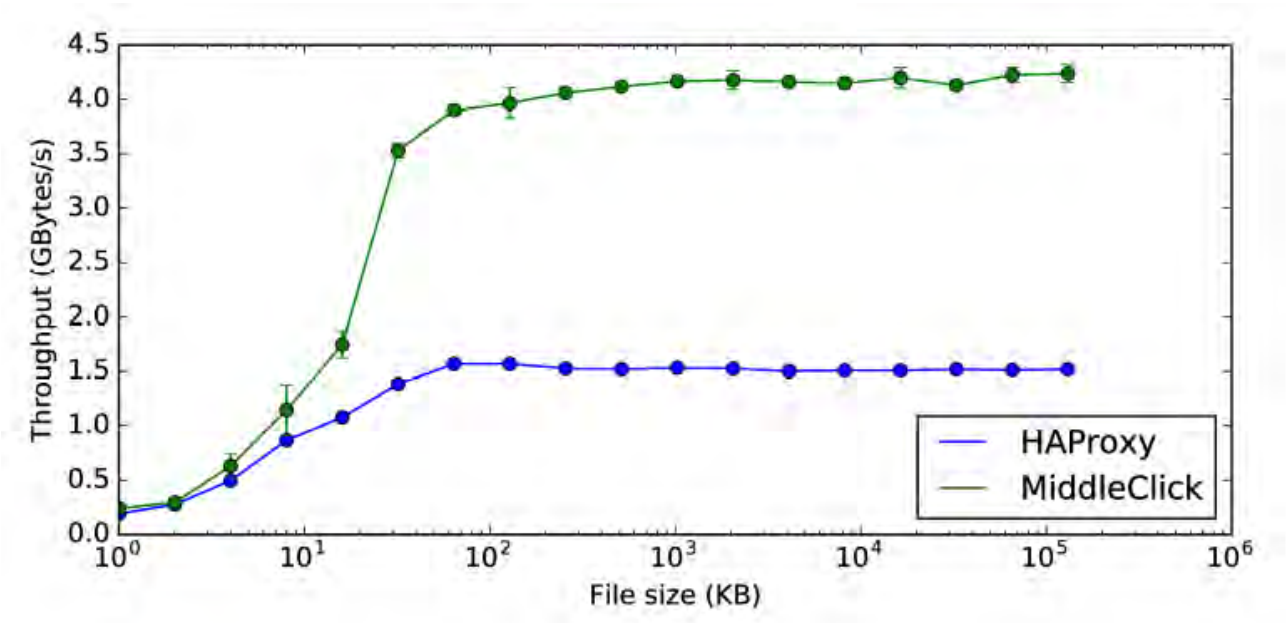


Figure 32: Data downloaded through a load-balancer using 128 concurrent connections. This is equivalent to 130 000 requests/s with 8-KiB files.

5.2 Heterogeneous processing

Although software packet processing performance improved dramatically, as illustrated by our own FastClick results, some packet processing tasks (*e.g.* classification, IP lookup) can still be done more efficiently on dedicated hardware.

One can of course combine software-based and hardware middleboxes in a service chain (*e.g.* offloading classification work to a SDN switch, as discussed in the previous section). However, in many situations, it is more efficient to use hardware co-processors directly attached to a commodity PC (*e.g.* GPUs, FPGAs, TCAMs, advanced NICs, network-processing boards, *etc.*). However, efficient communication between those devices is essential for the overall performance of the heterogeneous system.

As a first experiment, we developed such a fast communication mechanism to allow the communication of packets (or part of packets) between a commodity PC and a Tilera board (a network processing board with a many-core architecture, and embedded network interfaces). We managed to have a version of FastClick running on the Tilera, which allows to reuse most Click elements directly onto this board without any change in their code (contrarily to what would be needed for a FPGA or GPU, which both require special care). In the future, we will take advantage of specific network-oriented features of the Tilera, such as the MPipe architecture.


The communication between a FastClick instance running on the PC and one running on the Tilera is achieved through two **half-queue** elements. One element in the Click configuration on the PC, and one in the Click configuration of the Tilera. Internally, those elements transmits the packets through the PCIe bus. One can also choose to send only part of the packets (*e.g.* headers) to the other device.

The half-queue elements can intelligently batch small packets before transfer. While this requires more memory copying (and consequently more work on the CPU side), and incurs some additional latency, it diminishes the number of DMA requests, allowing for a better efficiency in some scenarios. Whether small packet batching improves the performance or not depends on the packet size distribution, and it is thus user-configurable.

The final achieved throughput is close to the PCIe bus bandwidth (the theoretical limit is 27 Gbps in our case), with 24 Gbps bandwidth for a real-world traffic trace captured at ULiège campus (containing, amongst others, both MTU-sized packets and small TCP ACKs, for an average packet size of about 1 kB).

Using those half-queue elements, we can now split a packet processing task between the CPUs and the Tilera cores, allowing us to explore which tasks are more efficiently done on one or the other platform. It also opens the way to implement some Tilera-optimised packet-processing elements.

For now, splitting a configuration between a host PC and a Tilera is done manually. But in the future, the resource allocator could do such splitting automatically according to application requirements and current resource use. Figure 33 illustrates how configuration splitting is done with half-queue elements. Packet processing elements B and C are moved to the Tilera, which is connected through half-queues *HQIn* and *HQOut*. The example assumes that the output port for C is irrelevant (*e.g.* if going to the same switch). Else, we would need another pair of half-queues to bring the packets back to the CPU side.

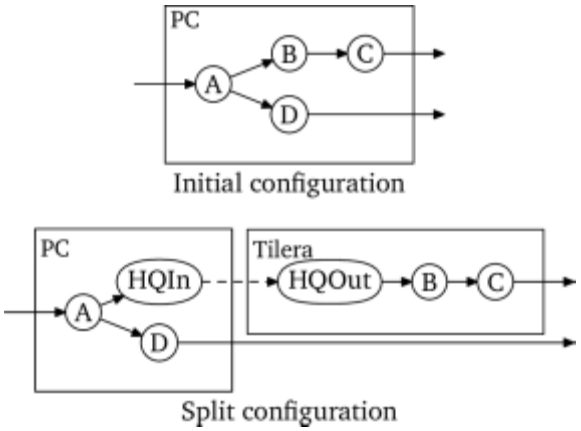


Figure 33: Example of configuration split for heterogeneous packet processing.


6 The Network Performance Framework

Network Performance Framework (NPF) [76] is an open-source framework we developed to allow to replay networking tests automatically from software. It is able to generate graphs, and comes with basic statistical analysis support.

This tool eases the automation of networking performance experiments with the aim of helping in the characterisation, modeling and optimisation of network functions placement and resource allocation, as will be explained in more detail in the next section.

NPF is based on comprehensive test description files called *testies*, which describe how to run a single test. A testie describes what software to run, how and where to run it.

Each test comes with a range of possible parameters (e.*g.* number of threads, buffers size, packet generator length or rate, ...), that NPF can use to compare performances (throughput, delay, ...) of multiple different programs and multiple different versions on a complete matrix of parameters (*i.e.* it will try all possible combinations of parameter values).

NPF can be used to try many different configurations and select the best ones according to multiple scenarios, using statistical tools to transform the result of the grid-search testing to human-understandable information such as the importance of a given variable. It does so using machine learning tools (using the scikit-learn toolkit), and can generate a visual regression tree to understand the big classes of performances and how to reach them, amongst other best/average/median/deviation values.

By sharing their testie files, researchers can provide a way to easily reproduce their results, and evaluate their robustness.

6.1 Comparison mode

Figure 34 shows a testie that will run *iperf* with and without the zero-copy parameter using 1 to 8 parallel connexions, and the generated graph. A graph is always automatically generated for each test, using line plots or bar plots, grouping variables when needed to make the results understandable as quickly as possible.

```
%info
Simple IPerf test

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

%script@server
iperf3 -s &> /dev/null

%script@client
iperf3 -c server -P $PARALLEL \
 $ZEROCOPY | tail -n 3 \
 | grep -ioE "[0-9.]+ kbits"
```

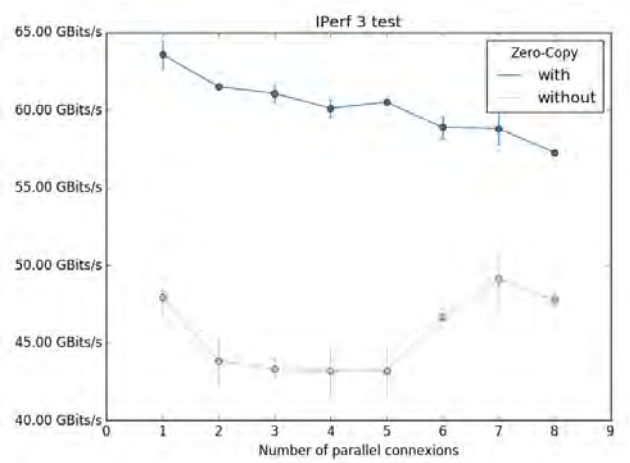


Figure 34: Evaluating the impact of zero-copy option of iperf.

`%info` describes the test, while the `%variables` section introduces the different parameters that will be varied between tests, and their ranges of values (here, 1 to 8 for `PARALLEL`, and nothing or "-Z" for `ZEROCOPY`).

`%script` describes the command to be run to launch the test. There can be multiple such directives for different machines, when the application is distributed, postfixed with target machine name. Here, one command is run on the iperf server, and one on the iperf client.

Figure 35 illustrates a second example that compares multiple implementations of the same function, *i.e.* a TCP traffic generator, using *iperf* and *netperf*. The `%script` directives are now prefixed with the tool name, in addition to the target machine postfix. The comparison tool is suited to compare multiple solutions and pick the one which perform bests under a given workload.

```
%variables
PARALLEL=[1*8]

%iperf:script@server
iperf3 -s &> /dev/null
%iperf:script@client
iperf3 -c server -P $PARALLEL -Z \
 | tail -n 3 \
 | grep -ioE "[0-9.]+ kbits"

%netperf:script@server
netserver -D -4 &> /dev/null

%netperf:script@client
echo "RESULT $(netperf -f kbits -l 2 -n
$PARALLEL -v 0 -P 0)kbits"
```

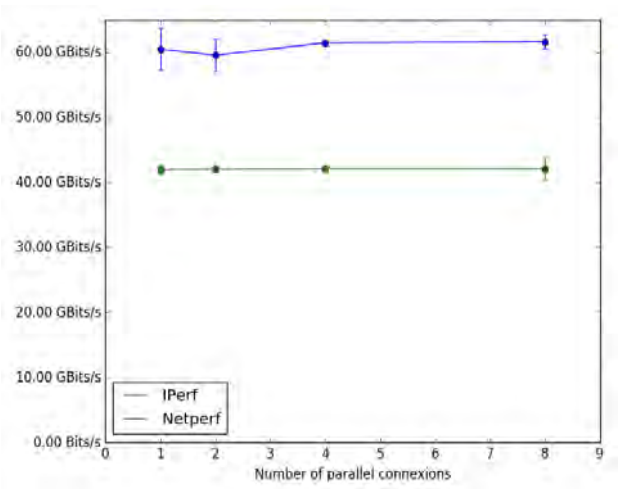


Figure 35: Comparing iperf and netperf.

6.2 Regression mode

The following example uses a Click configuration (we omit L2 advertisements for brevity) to build a packet generator. In the Click project, this configuration is often part of other testies to generate a specific traffic against a Device Under Test (DUT). As for any project under active development, regressions could be introduced. NPF comes with a regression library which will compare the performance results of multiple versions of the same software, ensuring that last versions or git commits did not break performances.

The `%config` directive specifies the software to use (see section 6.4). In this case, we need to build Click from source for the last 10 git commits.

```
%info DPDK L2 FastUDPGen + Fwd test

%config
require_tags={click}

%variables
LENGTH=[64*1024]

%script
click --dpdk -n 4 -c 0xf -- CONFIG

%file CONFIG
FastUDPFlows(RATE 0, LIMIT -1, LENGTH $LENGTH, SRCETH $MAC0, DSTETH $MAC1,
 SRCIP $IP0, DSTIP $IP1, FLOWS 1, FLOWSIZE 1000)
```


```
   -> uq :: Unqueue(32)
   -> ToDPDKDevice(1);
FromDPDKDevice(1)
   -> ac :: AverageCounter
   -> Discard;
DriverManager(wait 2s,
 print "RESULT $(add $(mul $(ac.byte_rate) 8) $(mul $(ac.count) 24))")
```

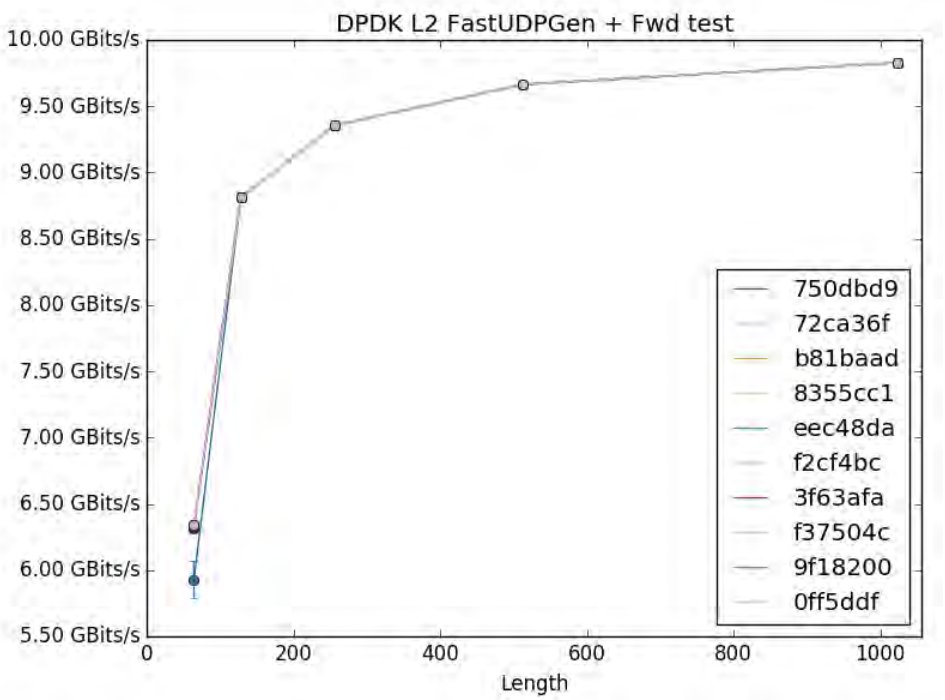


Figure 36: Click-based performance regression test. By default, the regression tool will test the last 10 git commits for git-managed software.

It also integrates a watcher tool to follow a git repository and re-run all testies upon new commit for continuous integration.

6.3 Statistical analysis

In a real-life scenario, the number of parameters can quickly grow, leading to results that are hard to graph and interpret. We use multiple statistical and machine learning tools to help understand the results. The following configuration shows an advanced version of the Click-based packet generator which pre-creates all packets in memory and replays them in loop.

```
%variables
BURST=[1*256]
LENGTH=[64*1500]
```


```
QUICK_CLONE={0,1}
STOP=[100000*51200000]

%file CONFIG
is:: FastTCPFlows(0, $BURST, $LENGTH, $MAC0, $IP0, $MAC1, $IP1, 5, 20)
-> MarkMACHeader
-> r :: MultiReplayUnqueue(STOP $STOP, QUICK_CLONE $QUICK_CLONE, ACTIVE false)
-> ac :: AverageCounter
-> Discard;

finish :: DriverManager( print "Launching test !",
 write r.active true,
 write ac.reset,
 wait 5s,
 print "RESULT $(add $(mul $(ac.byte_rate) 8) $(mul $(ac.count) 24))", stop);
```

The total number of parameters is combinatorial and, in this case, we end up with 900 possible combinations.

6.3.1 Best value

After executing all the (900) tests, NPF can output the best combination of parameter values, in our case:

```
BURST = 32, LENGTH = 1024, QUICK_CLONE = 1,
STOP = 25600000, 667014208784.33
```

6.3.2 Regression tree

But the tool does not stops there, it uses regression trees to compute the importance of each variable.

```
Features importance :
 BURST : 0.02
 LENGTH : 0.68
 QUICK_CLONE : 0.29
 STOP : 0.00
```

We see that the stop parameter is not influencing the performance in this case.

The regression tree can also be visualized. All leaves can be interpreted as classes of performance, and the value to reach them are always the one making the biggest differences.

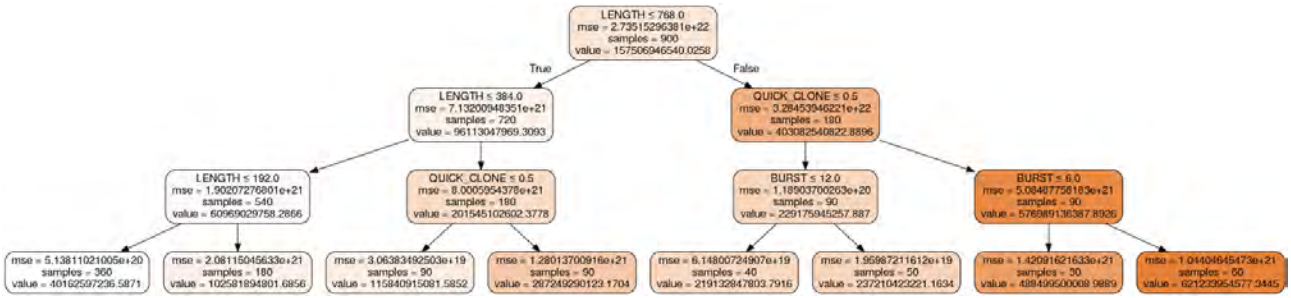


Figure 37: NPF-generated performance regression tree.

The root shows us that the length is the most important variable for throughput performance of the traffic generator, which makes sense as the performance cost is per-packet and not per-byte. We also see that the quick-clone parameter should be 1 and the burst size bigger than 8 for best performance.

The command line also allows to fix parameters to advance from deductions to deductions. In this example the whole left part of the tree gives no information as it separates different packet lengths. Considering only small and big packets (64 bytes *vs* 1024 bytes) would provide more interesting results. NPF includes a result cache so that reducing the matrix size is instantaneous.

6.4 Software definitions

NPF allows to specify the software to use in various ways:

- Built-in git support allows to go back in history to find previous regressions, or a baseline commit.
- Software can be fetched through HTTP.
- Software can be fetched through the OS package manager.

Here is an example that allows to download iperf and build it from source:

```
name=IPerf
method=get
url=https://iperf.fr/download/source/iperf-$version-source.tar.gz
version=3.1.3
tags=iperf
bin_folder=iperf-$version/src/
bin_name=iperf3
configure=cd iperf-$version && ./configure
make=cd iperf-$version && make
clean= cd iperf-$version && make clean
```


NPF also supports inheritance between configuration files, so that you can change only some configuration options without having to repeat a full configuration.

6.5 Conclusion

NPF allows to quickly explore the configuration space of networking applications in a repeatable manner. It can also use statistical analysis and machine learning tools to help to analyse performance results.

Although this work is still in-progress, we extended NPF with intelligent configuration space exploration, in order to avoid having to experiment all combinations of parameters, which does not scale well in larger applications. You will find more details about these extensions in section 7.3.

7 Joint modeling and optimization for function allocation

The architecture of Superfluidity is depicted in figure 38. The goal of this task is to optimise the allocation of network functions in the NFVI, *i.e.* map graphs of RFBs to available RFB Execution Environment (REE) resources, while meeting individual application SLAs.

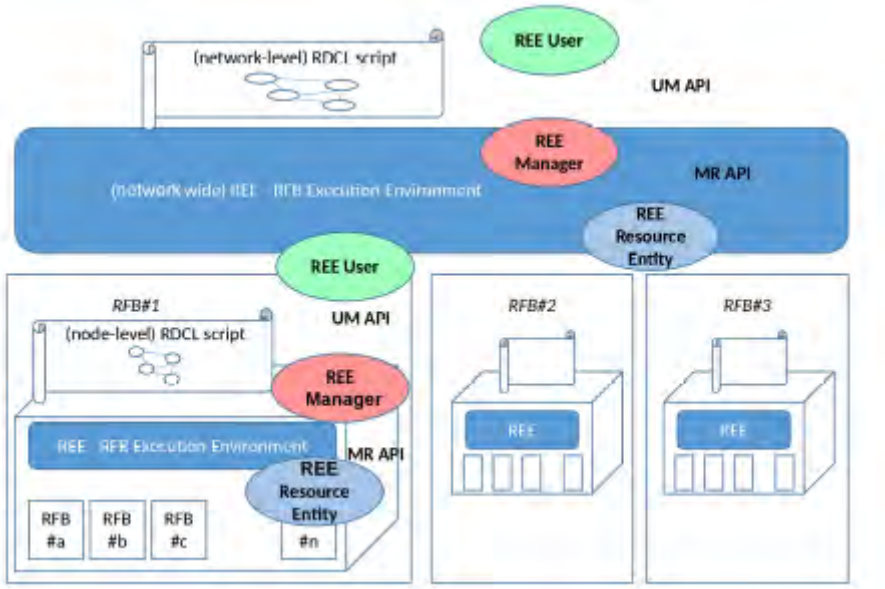


Figure 38: Superfluidity architecture, from D3.1.

In our initial plan, the function allocation optimisation was separated into two well distinct steps:

1. **Modelling** the system. For one given configuration, setup and workload, the generated model should allow to estimate relevant KPIs such as resulting bandwidth or latency.
2. Use the model for solving the **optimisation problem** of minimizing resource use, under the constraints that each individual SLA should be satisfied. The constraints could be relaxed a bit according to traffic conditions, *i.e.* we can temporarily violate SLAs, provided that we adapt fast enough to new traffic conditions, and that the violations don't occur too often (as was mentioned in the section 2 about utility functions).

However, after thorough characterisation of different resource environments (namely, VM-based clusters and Click-based machines), both in this task and in task 4.1 of WP4, it appears that this separation of concerns is problematic.


With many discrete interactions between various components, most of which behave in a very non-linear way (*e.g.* seeing a catastrophic collapse in performance once certain thresholds for some parameters are reached), it is hard to come up with good performance models, which would be at the same time accurate (*i.e.* making good predictions), human-understandable and relevant to solve the optimisation problem.

As we have seen with NPF (in section 6), the configuration search space size also grows exponentially with the number of nodes and parameter values, and quickly become intractable exhaustively for realistic deployments.

To alleviate those problems, rather than solving the modeling and optimisation problems separately, we propose to jointly solve them using **machine-learning** techniques.

The main idea is to train the system with experimental (or simulated, when possible) results, including :

- The ***logical configuration***, *i.e.* the input graph of reusable function blocks (RFBs) asked by the application;
- Information about the traffic **workload**, which can heavily influence the results;
- The **actual configuration**, *i.e.* the full details of how the RFBs were deployed in the infrastructure, with all the relevant parameters.
- A measure of the **performance** of the deployed application, gathered through telemetry or application goodput, and evaluated using utility functions.

Then, for a new logical configuration and workload (input features), the system would generate a set of possible actual configurations, along with their expected utility. We can then rank those configurations according to their utility, and choose the one with the best expected performance. Finally, once deployed, we measure the actual performance to close the feedback loop, using the new deployment as a new training sample, and adapting the configuration if needed (*i.e.* if the prediction was wrong) or desired (*e.g.* to try alternative configurations that might give even better performance, lower consumption of resources, or just to train the system further).

7.1 Challenges

Although machine learning has already been used to solve some resource allocation problems (*e.g.* [77]), it was generally in a much more restricted


context, and we need to overcome several challenges in order to come up with a good machine-learning based solution to our function allocation problem.

First and foremost, the configuration space is huge. One cannot just take all available input features and hope to learn something meaningful from them. We need some form of feature selection and pre-processing, which is far from trivial (more on this below). Secondly, there is an exponential number of possible configurations, of which we can explore only a very limited subset, so that we need to find smart ways to explore that search space.

One way to reduce the complexity is to use a layered, top-down approach. Such an approach has been chosen for the Superfluidity architecture (see figure 38), which is layered. For example, to deploy a network-wide function, one could first select on which nodes each high-level component should go. Then, each node would be responsible for its own resource allocation, maybe splitting the component into sub-components, to be further allocated to different sub-nodes (*e.g.* a Click graph would be assigned to a machine, which would then be responsible to assign the different Elements in the graph to different CPU cores, NIC queues, *etc.*).

The fact that the performance behaviour is non-linear, with many interactions between seemingly unrelated components, means that the information model should likely contain quite a large amount of information (*e.g.* a large number of layers and neurons per layer in a neuron network), which in turn requires a large training set. How we explore the search space is thus once again critical.

One way to get more training experiments is to vary the configurations in an online, reinforcement-based learning system. Once an actual configuration has been selected, and its performance measured, we can try close (for some to-be-determined notion of distance) variations of that configuration to obtain new samples. This, however, requires the ability to easily reconfigure the live system, without incurring any service interruption. We can apply a similar approach to short-lived, recurring applications. Rather than systematically trying the best estimated allocation, we would sometimes choose some different configuration, in order to improve the underlying model (without the need for live reconfiguration in this case).

Finally, the graph-structure of the configuration is hard to exploit by current machine-learning techniques. It is far from trivial to choose how we represent the configuration as a tabular input to be fed to the machine-learning algorithm. Should the graph structure be represented explicitly, or left non-observable? How to represent it? To illustrate the difficulty, consider the similar, but simpler problem of detecting whether or not a given object is present in an


image (in our case, we rather want to detect whether or not some pattern is present in the graph). Using the whole image as just a contiguous array of pixels is generally not efficient. Instead, we learn to recognise the object on small images containing just the object, and then use kernel-based techniques to pre-process the input image, splitting it into a large number of smaller sub-images, by sliding a window over it at various scales. The detection algorithm is then run on all the sub-images, and the object is detected in the larger picture if it is in one of the sub-images. We might need something similar in our case, but the way to decompose a large configuration into a set of smaller ones is not so clear. Moreover, detecting patterns is likely not enough to estimate the overall performance of a configuration, we might have to consider the global picture.

Encoding the structure of the RFB graph can be important for two reasons. First, for a given application, one *logical* configuration (describing just the intended semantics of the network function) can be implemented using a lot of different *actual* configurations, *i.e.* graphs of RFBs that encode not only the semantics but also the resource allocation. For example, elements and links might be duplicated for parallel processing on multiple cores, queues might be introduced to split a pipeline between several cores, *etc.* Examples of such encoding issues will be discussed further in section 7.3.

Another reason why the encoding of the structure matters is to allow *transfer learning* between different applications, *i.e.* being able to reuse a large part of a model learned on one application for another application. This would allow to learn a model offline based on some known network functions (this learning could be time and resource intensive), and then quickly adapt that model for the real-time deployment of novel, yet-unseen applications. This is often done in image recognition where a model trained on images of, say, cats and dogs, can be quickly re-trained to recognize cars and flowers. However, in that case, input features are always images of a given size. In our case, it is not trivial how to encode configurations for different network functions in such a way that we can easily reuse a learned model.

7.2 An open dataset for network function performance

To be able to answer the questions outlined in the previous section and assess the effectiveness of various machine-learning methods for the optimisation of resource allocation to network functions, we need to a dataset of network configurations, input traffic workload and corresponding measured output performance.


We identified that the question of how to use machine-learning for efficient resource allocation of network functions is of interest to other researchers. Moreover, we could not answer all aforementioned questions during the course of the project, and wanted to help foster future research into that direction.

Consequently, we decided to design, build and distribute an open dataset in the spirit of the OpenData initiative.

This dataset, built with the help of NPF (section 6), is constituted of the following features for a few networking applications:

- *Input traffic workload*. To have a common set of workload features across the different applications, we use the same traffic generator for all of them. We chose to use wrk in combination with nginx as a use case. By varying the number of concurrent threads, maximum number of concurrent connections and requested file size, we can control both the requested bandwidth and the length of the flows (small files give short-lived flows, while large files give long-lived flows). We also vary the number of destination server addresses to influence the self-similarity of flows (all addresses are mapped to the same server, but using several allows to study the impact of caching, *e.g.* when routing).
- *Measured output performance*. The wrk client reports goodput and delay. In addition, we measure link rates and energy consumption on the device-under-test (DUT), using smart plugs for the energy consumption (the CPU scaling governor is set to ondemand, and frequency is reset to minimum frequency between experiments).
- *DUT configuration*. These features are specific to each individual network function. Some are simple configuration parameters (*e.g.* the quantity of RAM devoted to packet buffers, queue and batch sizes, *etc.*), while the structure of the configuration graph itself is more challenging and is the focus of section 7.3.

The networking functions used for the DUT as of now are the following:

- A transparent *forwarder*, acting as a L2 bridge, which just forwards Ethernet frames between its two NICs. This application which does not even touch the packets is used as an absolute baseline (*i.e.* no network function can have better performance).
- A *switch*. As we statically fill ARP tables of the nodes before experiments, it is only different from the forwarder in that it rewrites source and destination MAC addresses in the Ethernet frames.


- A *router*, which does the same forwarding, but using L3 information (*i.e.* doing an IP lookup).
- A *network address translator* (NAT).
- A *firewall*. In this application, some features are related to the filtering ruleset (*e.g.* the number of filter rules).

We also considered more CPU-intensive applications such as and IDS, but we haven't yet implemented one of those.

7.3 Network configurations generation

The end-user only provides a **logical** configuration, *i.e.* a configuration that describes a single instance of a network function (NF), without deployment considerations such as batch size or thread allocation. The system must then be able to generate actual deployment configuration variants, for varying amounts of resources (mostly, allocated threads).

Many deployment variants can be generated by simply varying configuration variables such as batch size, or total number of memory buffers, using the `%variables` section of the testie file as explained in section 6. However, generating variants of the Click graph for thread allocation is more challenging.

As we want to investigate the performance benefits of various core allocation approaches, we don't just rely on the existing FastClick automatic resource allocation, but we explicitly assign Click elements to the available cores. Moreover, we need a way to encode the configuration in a tabular fashion, suitable for many machine-learning algorithms (rather than outputting the actual configuration graph itself as an input feature, which is unlikely to work well).

In the remainder of this section, we focus specifically on the problem of generating many graph variants of a given logical graph. It is decomposed into four steps:

1. **Generation.** The Click configuration is represented as a directed acyclic graph (DAG), where each element is a node, and each link an edge. Each edge is associated with a set of threads, which are the threads that will flow out of that edge. We also associate a virtual input edge to source elements, in order to account for threads to which that element is scheduled.
2. **Canonicalisation.** Of all the configurations that are isomorphic up to a reordering of threads, we only need to keep one of them. In practice, we


will try to avoid generating symmetric configurations during the generation phase, rather than filtering duplicates *a posteriori*. This might impact the learning process, but drastically reduces the number of experiments to run (*e.g.* there are 8! = 40320 ways to order 8 threads). One can always postprocess the data to de-canonicalise it if needed for learning (*i.e.* replicating the measures of one experiment for all other thread orderings).

3. **Filtering.** Many of the generated configurations are worthless and can be filtered out *a priori*:
    - Some generated configurations don't make sense. *E.g.* a path where input on thread 1 flows into output on threads 1 and 2 would introduce a `Pipeliner` on thread 2 (`Pipeliner` is an element that pass packets coming from one or more threads to another thread, using a queue). However, this `Pipeliner` would never receive any packet as only thread 1 is processing the input, and is pushing packets to output directly. In this case, the configuration would at best be equivalent to one were only thread 1 appears in the output, and could actually be worse (if the Pipeliner is scheduled, leading to context switches just to notice that its queue is empty). We can however find alternative ways to handle this specific problem, as will be explained below.
    - Configurations with a lot of core changes have a high cache miss cost, and a high scheduling overhead. We can thus safely discard configurations where the number of core changes on a path is too high, or where there are too many tasks per core.
4. **Instanciation.** Given the logical graph and a list of output threads for each edge, it remains to instantiate an actual configuration, which inserts `Pipeliners`, `LoadBalancers` and `CPUSwitches` as needed (more details below).

Once we have a structural variant of the Click graph, we can then expand it into multiple configurations by varying other variables such as batch size or number of memory buffers (by taking the cartesian product of the two sets of configuration parameters).

7.3.1 Generation and canonicalization

We first allocate one of the subsets of the set of threads to each link. To avoid generating symmetric configurations which are isomorphic up to a reordering


of the threads, we use the following ordering. A thread can only appear in the thread set of a link if all lower-ID threads were already used in one of the previously processed links. Else, we replace it with the thread ID just above maximum seen ID. *E.g.* if we have already seen sets `{0, 1, 2}` and `{0, 3}`, the next useable ID is 4 and thread set `{0, 5, 7}` would be replaced by `{0, 4, 5}` (making 6 the new next useable thread ID).

To represent thread sets, we use a compact integer representation where the `i`-th bit is 1 if and only if thread `#i` is part of the set. However, for learning purpose, we use one-hot encoding, *i.e.* there is one feature per link and per thread which is set to 1 if the thread is used on output of that link, and 0 otherwise.

Using this approach, the number of possible thread allocations varies as illustrated in table 4.

CORES / EDGES	2	3	4	5	6	7	8
2	6	19	48	109	234	487	996
3	18	131	692	3 173	13 590	56 263	228 984
4	54	915	10 304	97 165	842 778	7 018 279	57 280 308
5	162	6 403	154 340	3 004 357	52 914 854	887 859 591	14 545 797 224
6	486	44 819	2 314 448	93 082 541	3 331 061 962	112 656 933 479	3 705 591 846 148

Table 4: Number of thread allocations using generation method.

When there are only 2 edges (*i.e.* a single Click link, since source element will have an associated virtual input edge), we have the following analytical expression for the number of possible order-independent thread allocations: $2^{c+2} - 3c - 4$.

When there are more than 2 edges, we found no analytical expression, but one can observe that the number of possible non-symmetric thread allocations grows roughly by a factor of $2^c - 1$ for each additional edge.

Unfortunately, our current approach is not yet avoiding all symmetric configurations, *i.e.* some generated configurations are isomorphic up to a reordering of the threads. *E.g.* a two-link configuration with allocation `{0,1}` for the first link and `{1}` for the second is isomorphic to the configuration with allocation `{0,1}` for the first link and `{0}` for the second (renaming thread 0 thread 1 and vice-versa). We only detected this problem recently and are


working on an alternative approach. However, it won't change the fact that the number of potential configurations increases exponentially with the number of edges and cores.

To decrease the size of the search space, it is important to limit the number of edges. This can be easily done by using the recursive capacity of RFBs to limit the number of top-level elements to be considered. *E.g.* a router configuration will have a large number of elements, as Click elements each implement a single, simple functionality such as decreasing the TTL field value. However, we can group elements in different subsystems (*e.g.* handling ARP requests, generating ICMP messages, *etc.*). Allocating the thread by subsystem (*component class* in Click parlance, and higher-level RFB in Superfluidity parlance) instead of individual elements, we can drastically reduce the complexity. The performance might not be totally optimal, but it is likely to be good enough if the groupings of elements are chosen wisely (*e.g.* related elements in a straightforward pipeline will benefit from good caching properties).

Our implementation generates allocations as described above. It is an iterator that generate allocations one at a time, rather than trying to keep the whole set in memory, which would generally not fit. In practice, speed is still acceptable for small graphs and numbers of threads, since each allocation will give rise to an experiment which is much longer than generating the next allocation anyway.

If you need to sample allocations heavily, it would be more efficient to do it directly in the class. *E.g.* if our class generates 25 thousands allocations per second and you keep 1 allocation in a million, it would take 40 seconds to generate one allocation we keep. In this case, it would be more efficient to sample directly at the edge level (*i.e.* skipping some thread allocations for individual edges), but it is not clear what impact this would have on sampling quality, *i.e.* you might introduce some significant biases. Instead, it is always possible to do a random sampling directly. *I.e.* you generate thread allocations at random, filtering out those that do not follow our ordering constraint, and the other constraints described in the next section. Which method is more effective depends on how aggressively you want to sample the feature space.

Allocation sampling is not limited to random sampling, but can use more advanced strategies like the following lattice search, that we are currently investigating. The idea is to first gather a set of uniformly spaced or random samples from the feature space. Then the next sample to experiment is chosen to be at the middle of one of the hyper-volumes delimited by current samples


in the feature space. The hyper-volume which will be further explored is chosen according to a probability distribution related to the performance of the bounding samples (*i.e.* each hyper-volume is associated with an average of the performance of its bounds, and the better the average performance is, the more likely it will be chosen for further exploration). The used probability distribution can be tweaked to adapt the exploration vs exploitation trade-off.

7.3.2 Filtering

Once we generated one thread allocation for the logical Click graph (with added virtual input edges for source elements), we check whether or not we want to keep that allocation, based on a priori knowledge. This mechanism is pretty generic, one can plug any heuristic he likes in the process.

For our dataset, we chose to limit the number of core changes along any given path to 3. If packets need to hop from core to core, caching efficiency will drop and global performance will likely decrease.

We also filter out allocations in which several threads traverse an element that is not multithread-safe, and cannot be safely duplicated (*e.g.* because it needs an exclusive access to some resource).

Once an allocation is filtered out, we simply discard it. If we keep it, we then move to its instanciation, *i.e.* conversion to an actual Click configuration.

7.3.3 Instanciation

When instanciating an allocation, a first consideration is whether or not the involved elements are thread-safe. If they are, we can keep a single copy of each element, which is simply scheduled on several threads, as illustrated in figure 39. If the elements are not multi-thread safe, and support multiple independent instances, they can be duplicated once per thread as illustrated in figure 40. If a single-threaded element does not support multiple instances, then any allocation where it would be traversed by multiple threads would have been already filtered out. Finally, we can of course mix and match single-threaded and multithread-safe elements, as illustrated in figure 41, where the left-pointing triangle element is a `CPUSwitch`, which has one output per thread (*i.e.* in the example, thread 0 would use first output, thread 1 the second, and thread 2 the third).

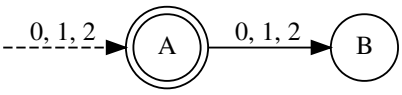


Figure 39: Instanciation of an allocation with multithread-safe elements.

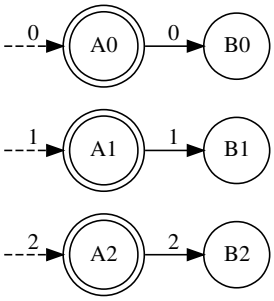


Figure 40: Instanciation of an allocation with single-threaded elements.

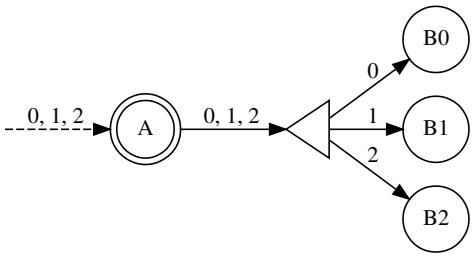


Figure 41: Instanciation of an allocation with mixed single-threaded and multithread-safe elements.

When output threads of a link are not the same as the input threads, we can use `Pipeliner` and `LoadBalancer` elements.

`Pipeliner` acts like a queue where multiple threads can push packets. It has only one output thread, which will pull packets from the queue and push it to the next element in the graph. If the thread that pushes a packet is also the output thread of a `Pipeliner`, it simply traverses the `Pipeliner`, there is no queuing in this case. *E.g.* figure 42 illustrates a configuration where element A is scheduled on threads 0 and 1, while the output thread on its link to B is assigned only to thread 1. Packets from thread 0 will be queued in the `Pipeliner` (represented as a queue) for later processing by thread 1, while packets coming from thread 1 in element A will pass through the `Pipeliner` directly to B.

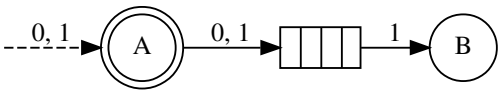


Figure 42: A change of core thanks to a `Pipeliner` element.

`LoadBalancer` will load-balance input packets (whatever the thread pushing the packet) onto several outputs. The output which is chosen depends on a hash of the packet fields (*e.g.* its RSS), so that packets of a same TCP flow will

always go to the same output. Note that, contrarily to `CPUSwitch`, all the outputs of a `LoadBalancer` can be used by all its input threads. *E.g.* figure 43 illustrate a configuration where element A receives packets on threads 0 and 1, which are then load-balanced for processing by B on threads 2 and 3, thanks to two `Pipeliner` elements and one `LoadBalancer` element (represented by a right-pointing triangle).

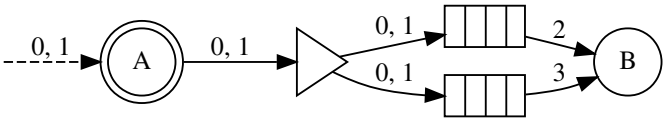


Figure 43: A LoadBalancer elements load-balance flows to several output threads.

A load-balancer is often needed when an output thread of a link does not correspond to an input thread, and some input threads are also present in the output threads. Consider the example of figure 44 where A fetches packets on thread 0, while B processes them on threads 0 and 1. Just inserting a `Pipeliner` is not enough, and results in an invalid configuration where the `Pipeliner` queue is always empty, starving thread 1 (as mentioned above, one could choose to filter out those configurations). However, we can also insert a `LoadBalancer` to obtain a valid configuration in this case.

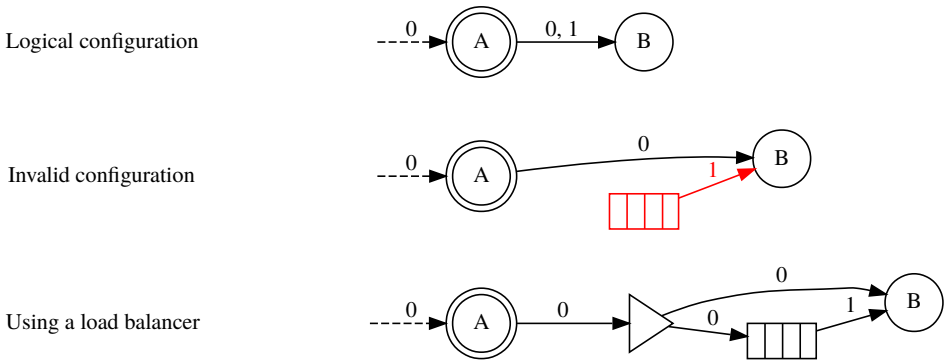


Figure 44: Not so simple instanciation of a simple allocation.

To get a feeling of how the instanciation of thread allocation proceeds, have a look at figure 45, which shows the 19 allocations generated for a single link between two elements A (source) and B (sink), assuming both A and B are multithread-safe.

Note that our current approach to try to avoid generating allocations that are isomorphic up to a reordering of the threads is not fully effective. While we pass from 49 allocations to 19, some configurations are still symmetric (*e.g.* allocations {{0,1,1}, {0,0,1}} and {{0,1,1}, {0,1,0}} are the same if you


rename thread 0 thread 1 and vice-versa). We could actually go down to only 12 order-independent configurations in this case.

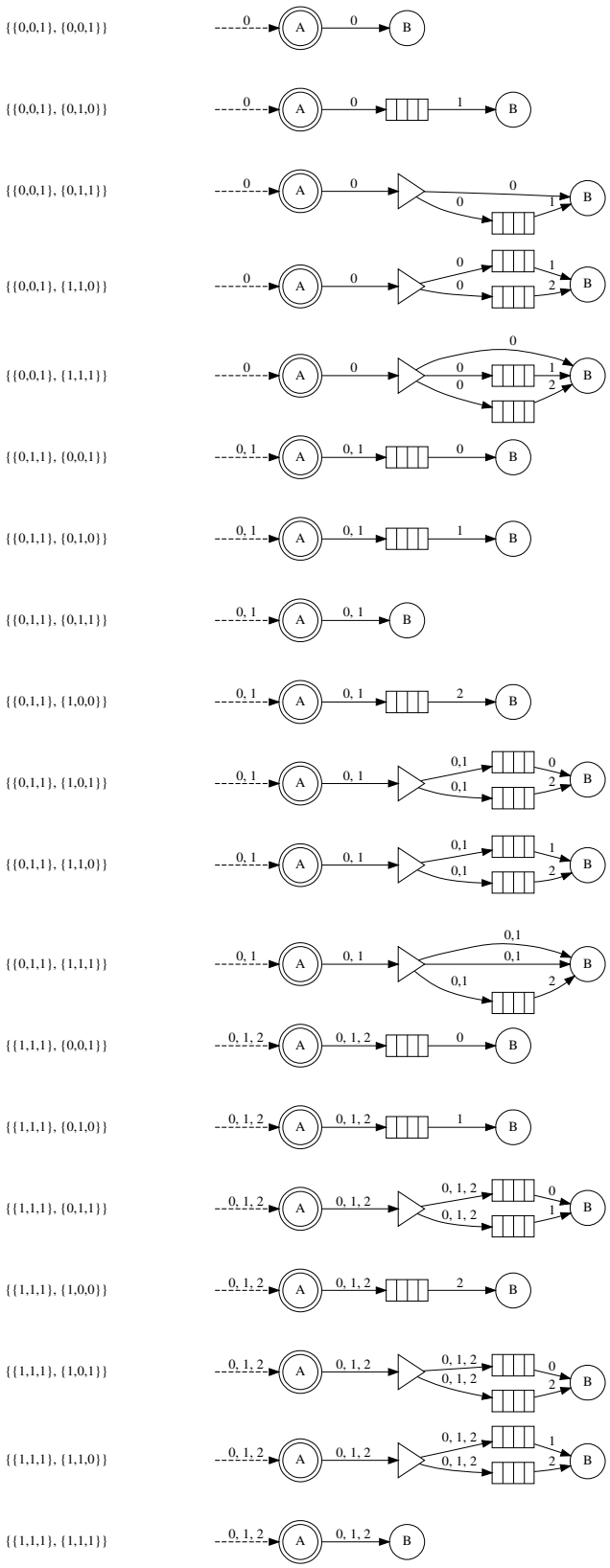


Figure 45: Instanciation of a single edge graph. Triangles represents load balancers, and queues represent Pipeliner elements.


7.4 Preliminary results

In this section, we will not do a complete evaluation of the aforementioned network functions using FastClick, as the dataset generation is not fully complete at the time of writing.

Instead, we will give an example using a simpler parameterisation (*i.e.* without taking the graph structure into account) of a VM-based firewall, using DPDK for packet I/O. Among the parameters in this experimental dataset are:

- the number of cores allocated for processing;
- the size of the DPDK pool;
- the size of the bursts, *i.e.* batch size;
- the sizes of the reception and transmission queues;
- the size of the ruleset (generated using ClassBench).

We considered two kinds of traffic workload: one using synthetic random traffic, and one replaying real traffic traces. The observed performance metric is the number of packets processed per second.

The dataset was analysed using Gradient Boost Regression Trees (GBRT) as the learning algorithm. This algorithm combines a set of *weak* learners such as decision trees to form a *strong* learner that can be used to solve classification and regression problems (see [78] for details). The parameters of the algorithm are given in table 5.

ESTIMATORS	LEARNING RATE	MAX TREE DEPTH	LOSS FUNCTION
500	0.01	4	Least squares

Table 5: GBRT parameters.

Figure 46 illustrates the case where the traffic is generated synthetically at random. The left-hand side of the figure illustrate the residual error as a function of the number of boosting iterations, while the right-hand side indicates the relative importance of features.

It shows that we can reach good accuracy in the prediction of the packet processing rate, and that the most important feature is the number of cores allocated to the application. The second most important feature is the size of the DPDK pool, and this one is counter-intuitive when you look into the numbers. The performance is actually better with less memory allocated, due to caching issues. As the same buffers are reused over and over, they can stay

in caches with a small pool size, while a larger pool size leads to more cache misses. Of course, allocating too little memory also degrades performance, which shows the interest of an automated approach to find the optimum, rather than relying on intuition or trial and error.

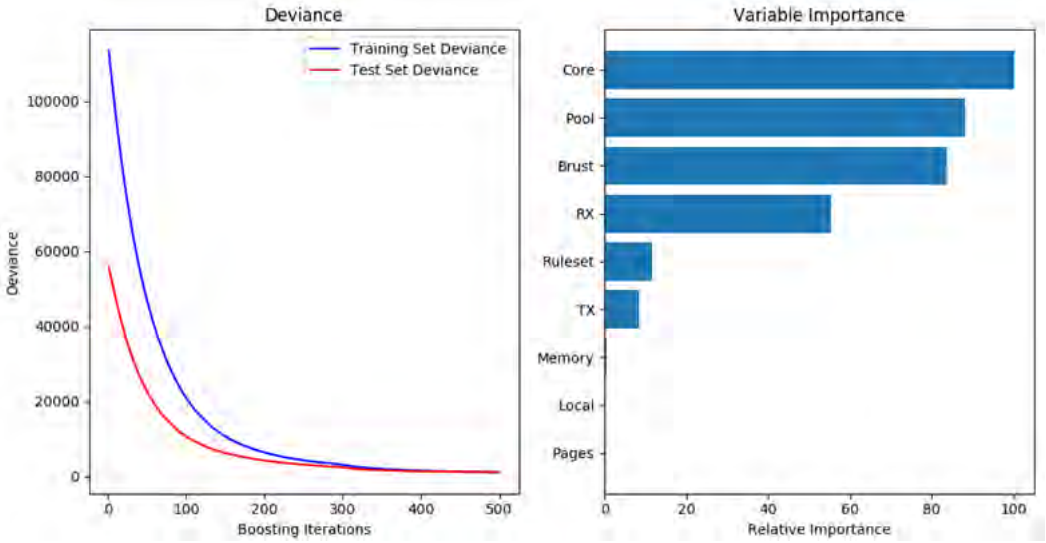


Figure 46: Firewall GBRT model on random traffic.

Figure 47 shows the same analysis for real traffic. We observe that the most important feature is no longer the number of allocated cores, but the burst size. This is because the real traffic contains high locality, with many consecutive packets being part of the same flow, and RSS will dispatch them to the same core. Moreover, it also leads to less cache misses, and the overall performance is better than on the random traffic.

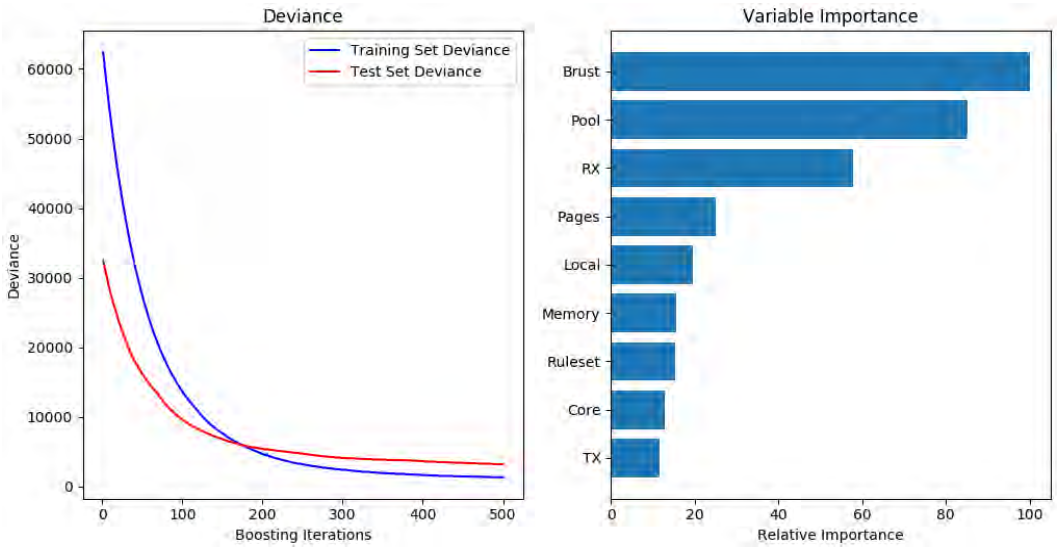


Figure 47: Firewall GBRT model on real traffic.


When we compare the two settings (random and real traffic), we understand the importance of the traffic workload shape, and the value of having an automated procedure to compute optimal parameters for specific traffic conditions.

We observed similar results on a Click-based configuration of a router. The prediction accuracy is good provided you consider enough samples, and the optimal parameters vary with the traffic conditions. It know remains to characterize exactly how many samples are needed for good prediction, how to choose them, and how we could reuse part of the model across different network functions.

7.5 Deployment strategies

How this work can be used for efficient network function placement and resource allocation depends largely on the answer of many still open questions as outlined in section 7.1. However, we can already describe some tentative deployment strategies.

7.5.1 Offline-learning for single applications

If a network function can be characterized offline (using synthetic or generated traffic), then we can learn a predictive model that, given the shape of the traffic workload and resources allocated to the application, answers with an estimate of performance metrics (in our dataset: bandwidth, latency and energy consumption) for the given configuration.

The limitation here is that the more complex the network function, the higher the number of required experiments to learn the model. In practice, however, we can simplify complex applications by using a recursive decomposition as explained in section 7.3.1. This is a trade-off, as we can miss the optimal configuration by simplifying it, but it is still much better than current approaches based on over-provisioned VMs. We can also trade-off the number of required experiments and the accuracy of the model, but the exact implication of this trade-off will vary with the learning algorithm used.

Once equipped with a predictive model, we can use the MAUT introduced in section 2 to provide an objective function, and apply various oracle-based optimization techniques to find a good configuration.

An alternative, simpler approach, is to use information about the NFVI to propose a few possible deployments, and then use the predictive model and MAUT to rank them, allowing to choose the one giving the best predicted utility.


Another limitation of this approach is that, as the model is only valid for one application in isolation, *i.e.* resources must be exclusively allocated to it, no sharing is possible (*e.g.* you cannot allocate half a CPU core, the other being use by another application, as this was not evaluated in the model, and has been shown to have effects on performance that are hard to predict).

To alleviate that, one could try to learn a predictive model for a full service chain, but we then bump into the complexity vs number-of-experiments trade-off again. The ideal solution to that problem would be to have a multi-application model able to predict performance for large new applications, but it remains to see if this is possible.

7.5.2 Online reinforcement learning

One way to gather many additional experimental results is to close the feedback loop after deployment, by leveraging the monitoring infrastructure to gather the actual performance metrics of the deployed configuration. Using some form of reinforcement learning, we can then update and enrich the model. When you have to deploy a new instance of a network function, you can use the current model to predict what the best configuration would be, but you can also try an alternative configuration to better enrich the model (*e.g.* using the lattice search approach outlined above). Of course, you might bias the exploration vs exploitation trade-off more towards the exploitation side, for production environments (*i.e.* you are willing to try a sub-optimal configuration, but still an acceptably good one).

7.6 Conclusions

Using machine-learning to solve the resource allocation problems in network function placement brings many questions. To answer those questions, we created an open dataset that maps traffic workload, resource allocation and measured performance for a few network functions. We hope that the release of this dataset as open data will help foster research on that topic, and we have already have had marks of interest, both in the networking and in the machine-learning communities. The creation of the dataset itself was far from trivial, with a combinatorial search space, and hard to exploit and encode graph structure.

The results gathered so far bring useful insights into the performance of network functions on commodity hardware, both for Click-based and VM-based environments. We have shown that, provided you run enough experiments, you can learn a good predictive model for some network functions. Such a model


can be used to predict deployments that will give the best utility to the cloud operator and its clients.


8 Performance optimization to the MicroVisor virtualization platform

A number of user-space I/O performance optimizations have been discussed previously. In order to further optimize the performance across the whole stack, we have developed some optimization techniques additionally in the virtualization layer. The MicroVisor is a distributed, light-weight hypervisor platform that has been designed with performance and scalability as two guiding attributes. It is derived from the commonly used Xen platform but removes the localised control domain (Dom0) on every node and instead centralises the control and command logic across a clustered set of MicroVisors.

The design and implementation of the MicroVisor is not carried out under the Superfluidity project, however the platform has been optimised and adapted to provide better performance for the Superfluidity usecase workloads. The MicroVisor exposes fine-grained hardware primitives along with acceleration features of the underlying hardware up through to the virtualised OS and the applications running on them. This fine-grained mapping enables NUMA awareness for guest vCPUs and physical memory alignment, as well as NUMA aware CPU pinning for guests. Some performance optimization features that are provided by the MicroVisor platform and are appropriate to the Superfluidity usecase workloads include:

- Virtual CPU to Physical CPU pinning.
- Separate scheduler policies (Credit, Real-time etc..) per subset of cores.
- CPU pool per NUMA node.
- Memory NUMA awareness at the hypervisor level for vCPU scheduling over physical cores.
- Alignment of a VM with its vCPU cores onto a particular NUMA set with optimised storage or network controller traffic handling based on physical hardware routing to PCI bus and the CPU socket.
- Exposing block-level storage devices to the physical network via ATA over Ethernet to accelerate remotely accessible storage (ATA-over-Ethernet block device backend in the MicroVisor layer to allow block requests over Ethernet).
- ATA over Ethernet – kernel module running in storage node VM as a server removes TCP protocol overhead.

- End-to-end support for virtual network packet fragmentation including TSO/GSO.
- Round-robin over Virtual NIC queues for sending data from the MicroVisor to a VM which is multi-queue capable.
- Pinning of Netfront and Blkfront queues to particular vCPUs within the storage node and driver domain.
- Integration of network device drivers directly into the Type1 hypervisor layer for certain hardware types provides the lowest access latency and packet forwarding delay on any commodity hypervisor platform.
- Link aggregation between MicroVisor nodes using the underlying physical network configuration options (striping, load-balancing, high-availability, daisy-chain).
- Virtual network overlays over the available physical network topology instead of layer 3 routed encapsulation techniques such as VxLAN.
- Lightweight Network Functions for packet processing, routing, encryption as unikernel instances, able to instantiate anywhere across the infrastructure with minimum startup time or service downtime.

Packet forwarding latency is a critical metric for performance, but is a hard metric to optimise since adding more powerful processors with more cores to the task does not help to improve the behavior. Therefore, every microsecond that can be saved is extremely valuable. The network packet forwarding latency from a VM to another VM running on the same physical host is measured and illustrated in figure 48. The latency on a MicroVisor platform is reduced by more than 50% compared to a standard KVM host.

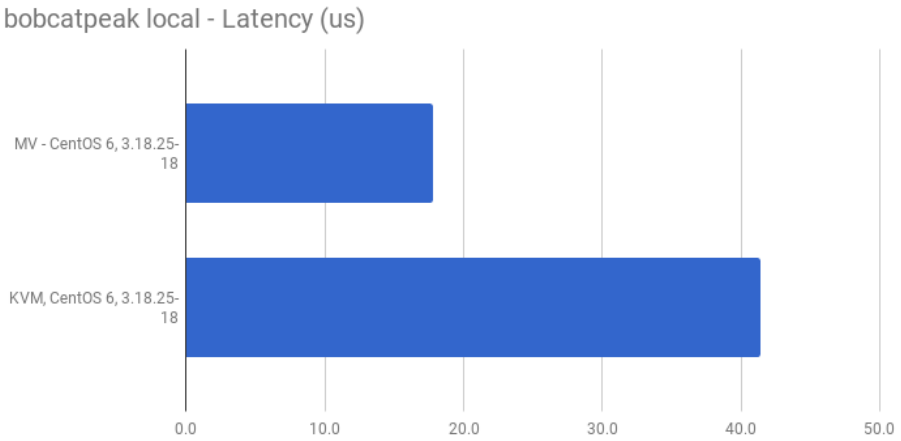


Figure 48: Network latency between VMs running on the same physical host.

The figure 49 shows the remote latency for VMs running across different physical hypervisors involving an external network switch. The MicroVisor platform is evaluated against bare-metal hosts and KVM hosts. There is

performance decrease on virtualised platforms like MicroVisor and KVM but MicroVisor outperforms KVM largely by more than 30%.

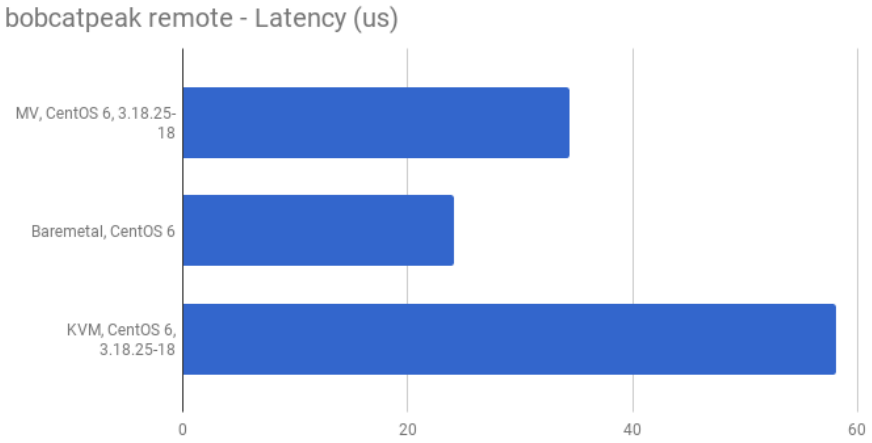


Figure 49: Network latency between VMs running across different physical hosts.

Regarding network traffic throughput both for a single path and for aggregate paths, the MicroVisor also significantly outperforms stock KVM. The throughput between local VMs running on the same physical host is showed in figure 50.

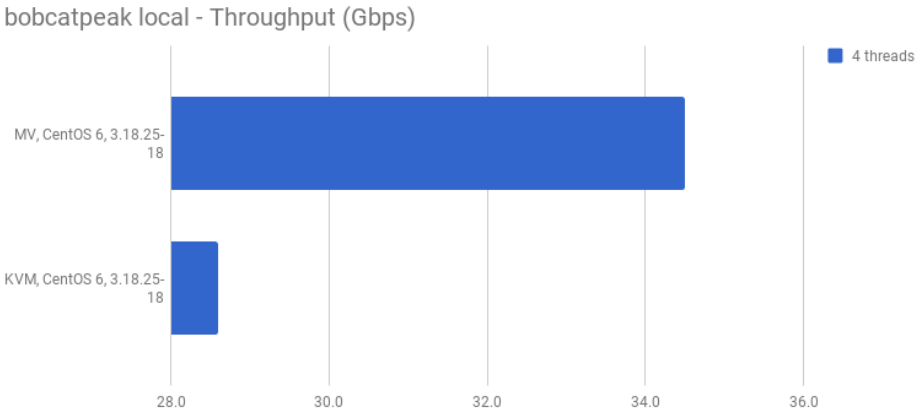


Figure 50: Throughput between local VMs running on the same physical host.

The network traffic throughput of the path aggregation technology loaded on MicroVisor platform is illustrated in figure 51. Our assessment demonstrates that end-to-end performance can scale linearly with the number of physical paths without depending on any complex switch bonding protocols.

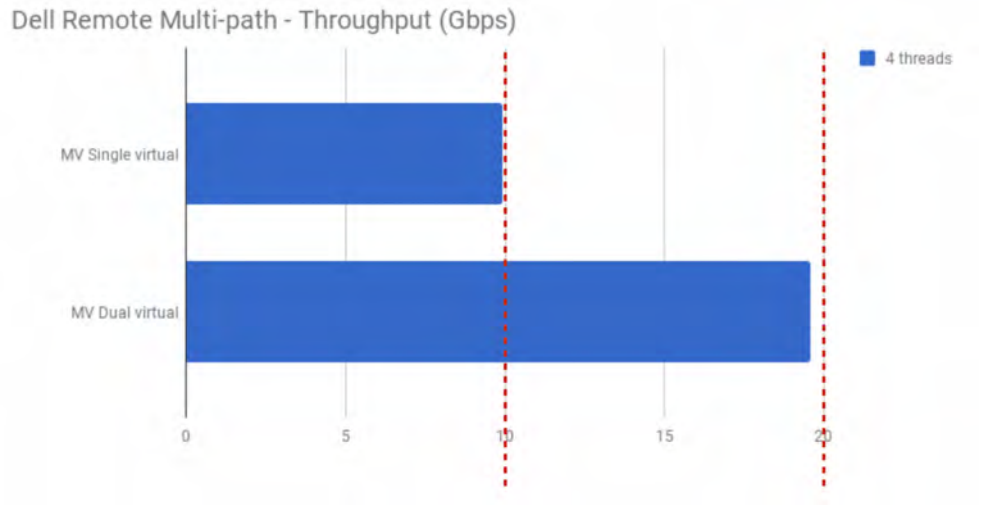


Figure 51: Network traffic throughput of the path aggregation technology.


9 Dynamic Resource Allocation for Guaranteed Service in CRAN Baseband Computing Platform

9.1 Introduction

In general, the CRAN computing system allows the virtualization of radio access protocols, network functions, edge services and CRAN management on a common hardware platform comprising general purpose and application specific processors complemented by dedicated accelerators and peripherals. While the virtualization of RAN management, edge services, network functions and control plane of access protocols is the state-of-the-art approach in RAN/CRAN design, virtualization of physical layer and baseband signal processing algorithms suffers from the relatively low computation capabilities of COTS server hardware and its unpredictable behaviour. However, the modern processors and SoCs (*e.g.* Intel's Xeon CPUs + Altera FPGA, Xilinx's Zynq ARM CPUs + Xilinx FPGA) tightly integrate a general purpose computing resources with embedded FPGA fabric, hence, allowing to couple CPUs with specific accelerators for computationally intensive and time critical tasks. The main challenges in this regards are in i) the provisioning of scalable heterogeneous many-core solution with flexible interconnect and ii) the mapping of dataflow baseband applications with specific communication patterns to underlying hardware (HW) resources while guaranteeing the determinism and real-time constraints. In the deliverable D5.2, we investigate scheduling and mapping strategies to assign the task of dataflow baseband applications to processing elements of heterogeneous many-core platform. In contrast to this, this deliverable addresses the problem of optimum allocation of resources in order to provide guaranteed service (GS) in baseband signal processing. More specifically, we address GS in terms of guaranteed bandwidth and latency of critical traffic in order to improve the responsiveness and determinism of application execution.

9.2 Resource allocation of dataflow-engine components

The concept of Superfluidity baseband signal processing platform is illustrated in figure 52. The compute node comprises general purpose CPUs with tightly coupled reconfigurable accelerators (ACC). In order to enable the performance scalability, a network on chip (NoC) with GS capability is assumed to interconnect CPUs, ACCs and memory subsystem. Multiple compute nodes can compose a compute cluster. Various compute cluster arrangements are

possible ranging from COTS servers with FPGA based ACCs to application specific MPSoC clusters (*e.g.* Tomahawk-4 MPSoC cluster illustrated in figure 52 have been assumed for 5G mobile broadband scenarios [44]).

The baseband signal processing application is represented by a dataflow graph that is mapped to HW resources by dataflow engine (DFE) in figure 53. DFE supports simultaneous execution of multiple baseband applications on shared hardware resources and allows the adaptation to various HW platforms according to workload conditions. The main challenge in this regards is to provide adequate computation power for signal and data-intensive tasks while meeting strict real-time requirements.

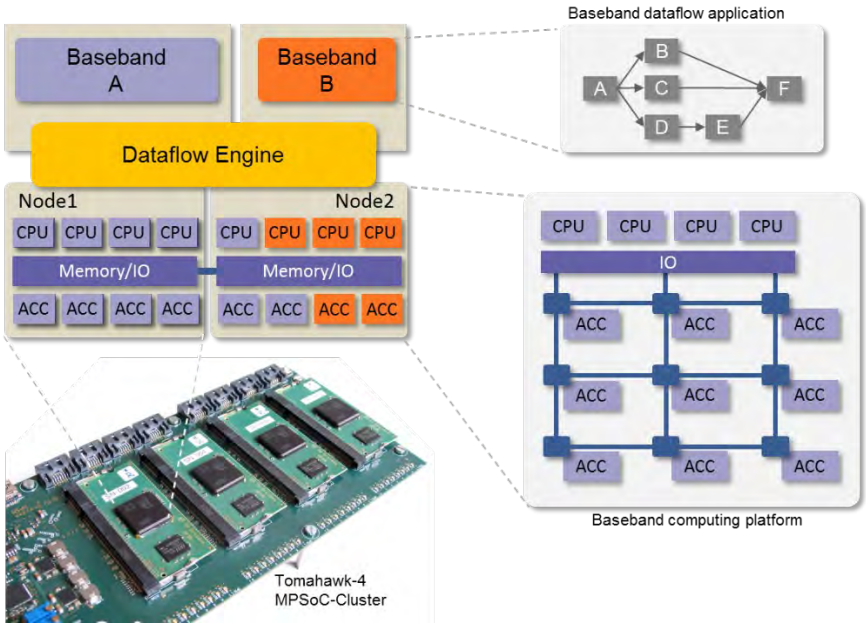


Figure 52: Concept of Superfluidity baseband signal processing platform.

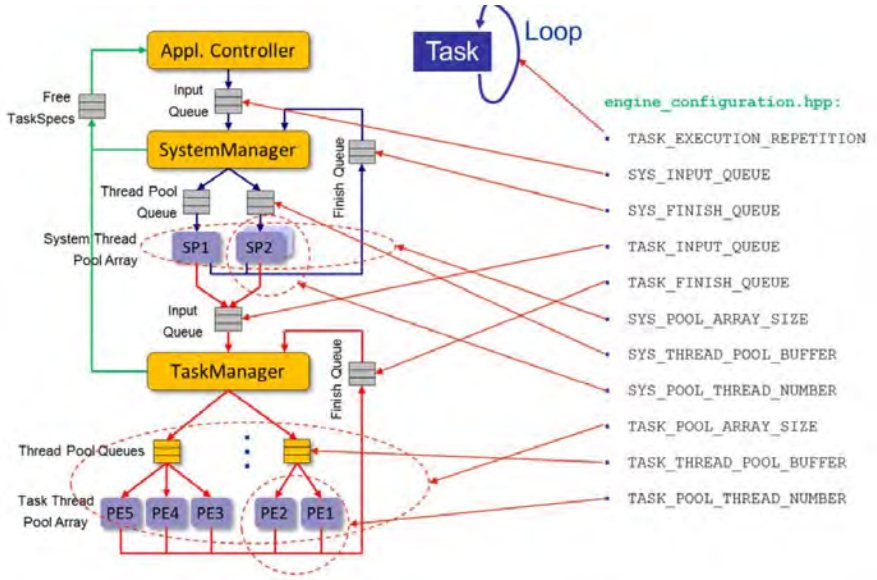


Figure 53: The architecture of dataflow engine and its configuration capability.

DFE architecture is highly flexible and allows engine configuration in terms of the number of logical processing elements (PE). The affinity of logical PEs to physical CPUs/ACCs can be defined statically or dynamically, which enables the optimization of system performance and efficiency. The following code segment illustrates the list of parameters driving PE thread affinity to physical CPUs:

```
// Thread Name CPU ID
const size_t THRD_MAIN_CPU = 4; // Main thread
const size_t THRD_TOP_CPU = 5;  // Application controller
const size_t THRD_SYSMNGR_CPU = 6;    // System manager
const size_t THRD_TASKMNGR_CPU = 7;   // Task manager
const size_t THRD_SYSPOOL_CPU = 8;    // Start of System pool
```

In this mapping example, one-to-one mapping of threads to CPUs is employed *i.e.* Main, Appl.Controller, SystemManager, TaskManager, SystemPool and TaskPool are mapped to dedicated CPUs.

9.3 Dynamic resource allocation for guaranteed service in multi-processor networks for baseband signal processing

Providing Guaranteed Services (GS) in terms of bounded latency and bandwidth in network is crucial for design of predictable systems. Circuit Switching (CS) is the frequently adopted technique enabling GS, which first allocates exclusive channels to form a circuit, *i.e.* the connection from source to destination, followed by sending data along the established connection. However, since the resource is exclusively occupied during the entire lifetime of the connection, it may lead to considerable system inefficiencies due to the blocking of resource for other traffic flows. In order to improve the resource

utilization, two extensions of CS have been introduced: i) Time-Division-Multiplexing (TDM) and ii) Space-Division-Multiplexing (SDM). In TDM CS, the link capacity is split into multiple time slots, and a subset of link time slots is allocated to a specific connection according to the bandwidth requirements as *e.g.* adopted in parallel probe search [45], AEthereal [46][47] etc. Analogically, the link is composed of multiple physical wires in SDM, and subset of the link wires is exclusively allocated to a specific connection.

Limited amount of TDM/SDM NoC resources poses a challenge to their efficient usage while ensuring the provisioning of requested capacity for all flows. This becomes particularly critical for highly loaded networks. In addition to this, the resource allocation problem has exponential complexity with the path length, which makes it difficult or even impossible for large-scale networks or dynamically reconfigurable application scenarios. This calls for powerful and low-complexity connection allocation methods that comprise i) fast and efficient selection of a contention-free path between source-destination pair and ii) allocation of the appropriate slots and corresponding resources on the path. However, recently proposed allocation approaches either suffer from long allocation time (software search approach, unidirectional search) or limited allocation success rate (supports only minimal path search) or limited scalability. This work tackles this problem by proposing a dedicated connection allocation unit - the NoC Manager (NoCM). The principle of connection allocation is illustrated in figure 54. Note that we assume the regular mesh network model in our investigations. However, the proposed approach can be easily adopted to any network topology.

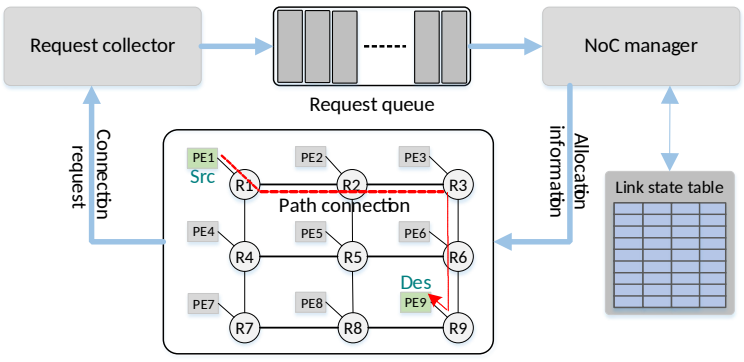


Figure 54: The principle of connection allocation in CS Network using centralized NoC-Manager approach.

The NoCM employs a novel trellis-search-algorithm (TESSA) that solves the allocation optimization problem with linear complexity by making use of dynamic programming approach [51]. This enables to explore all possible paths between source-destination node pairs within a guaranteed low latency. The


found path is ensured to be the contention-free shortest available path. Three different trellis structures are proposed and analysed for the purpose of different application scenarios: unfolded, folded and bidirectional. Moreover, the novel search approach in conjunction with efficient hardware implementation of NoCM reasonably improves the overall performance against recently proposed methods. Furthermore, the mathematical formulation of the path search is proposed, which allows a variety of criteria to be applied to obtain the global optimal results.

The allocation techniques can be grouped into two categories: i) static (design-time) allocation [53][54][55] and ii) dynamic (run-time) allocation. Since the static allocation is done at the design time and cannot be changed according to the applications' requirements during run time, they are not well suited for dynamic systems. The dynamic connection allocation techniques can be divided into two categories: i) centralized [51][52] and ii) distributed allocation [45][46].

In the centralized allocation scheme, a central manager is responsible for the connection allocation. Since the central manager has the global knowledge of the system, it can achieve global optimal results. The centralized allocations are typically based on software solution *e.g.* utilize x86, Microblaze or ARM processors. Software solutions provide excellent flexibility, however, they suffer from excessively long allocation time to support real-time systems. For instance, single path exhaustive path-search requires thousands of processor cycles for a single allocation. To increase the allocation speed, HArdware Graph Array (HAGAR) approaches were proposed in [56], in which a dedicated hardware connection allocator is used to speedup the allocation by two orders of magnitude against software methods. In HAGAR, the connection allocation problem is solved as a shortest path problem in a graph representation of the NoC. However, HAGAR is employed for basic CS, and does not support link sharing techniques such as TDM and SDM. In this work, we propose a dedicated allocator, NoCManager (NoCM) employing the Trellis search based allocation algorithm (TESSA) for TDM NoCs. The trellis search can search non-minimal as well as minimal paths, and can find the desired shortest path with a guaranteed low latency.

9.3.1 System model

The principle of connection allocation procedure using dedicated allocator NoCM is illustrated in figure 55. The NoCM attempts to allocate the appropriate connections when it receives connection requests. The NoCManager solves the

shortest path problem in a trellis graph description of the NoC in order to find the shortest free path and allocate slots between source-destination nodes. A block diagram of the NoCM is shown in figure 56 and comprises a trellis path search module and link state memory. NoCM collects and processes the incoming connection requests within the request queue. The resulting allocation parameters are sent through the NoC to respective source node in order to setup the connection.

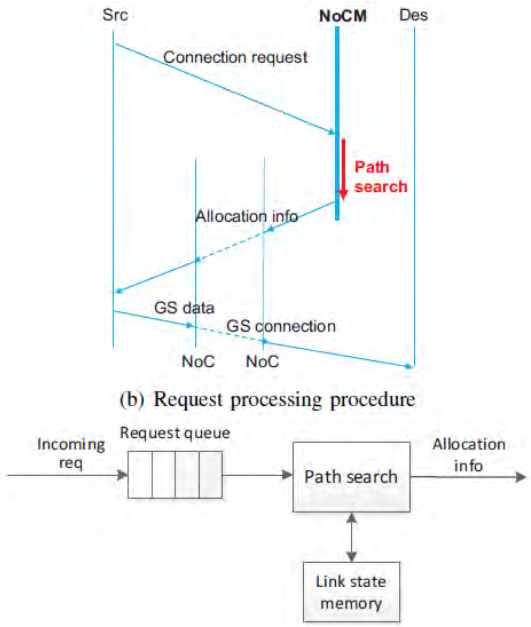


Figure 55: Connection request procedure for guaranteed service in NoC and the block scheme of centralized connection allocator.

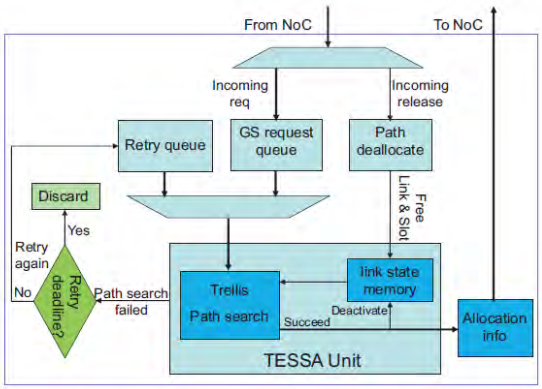


Figure 56: Block diagram of centralized connection allocator.

The aforementioned shortest path search problem has exponential complexity with the length of the paths, and the exact complexity function depends on the network topology. For a mesh network, if non-minimal paths are allowed, at each hop we have the choice of going in 4 directions. Assume l is the distance


between source and destination and d is the number of allowed detours, then the approximate complexity function would be O(4l+d). This problem can be efficiently solved by dynamic programming - an optimization approach that breaks down the complex problem into a sequence of simpler problems which are solved stage by stage, thus reducing the computation complexity to linear. This work is motivated by the Viterbi algorithm, which uses the dynamic programming approach for sequence estimation in the communication domain. We adapt the principles of Viterbi algorithm for efficient path search in NoC, called trellis path search algorithm. The successive path traversal from the source to the destination during the path search is represented by trellis graph. Thus the trellis graph is a time indexed version of the NoC graph (figure 57). There are five most important characteristics of the trellis graph:

1. *Stages*: The network traversal mapped on to trellis graph is structured into multiple stages (figure 57), which is solved sequentially stage by stage. The stage (*i.e.* the column) of the trellis graph contains all the nodes of the network and represents a single hop traversal through the network. We refer to the “decision stages”, referring to the number of stages which have to be traversed to make a decision, excluding the first stage, since the first or starting stage does not require any decision making. By default, the number of the “decision stages” is *2N-2* for *NxN* mesh network, which is equal to the longest minimal path (*i.e.* the longest possible path of all minimal paths) in the network. The stages have time implications to represent different time slots associated with different hops.
2. *States*: Each node of the trellis graph is called a state, and summarizes the knowledge in order to make the current decisions. At each stage, the decision in a particular state is determined simply by choosing one and only one of the active incoming branches as survivor path.
3. *State transitions*: The forward progress from one allowable state at a stage to another allowable state at the next stage in one unit of time (*i.e.* a time slot), is called a state transition. In the trellis graph, this is represented by a directed edge (or branch) connecting the two states. The state transition acts as the link between two respective routers corresponding to a forward hop in the network towards the destination.
4. *Branch metric*: The branch metric (B) is a measure of the transition that reflects the value (importance) of the branch. It is a function of several variables, such as the available slots of the branch (a), the number of requested slots (r), and the weight (w) that reflects the priority of the branch, *etc.* The information of available number of slots and the requested number of slots can be used to balance network load. The fewer the

available slots the branch can provide and the more the number of slots are requested, the larger the branch metric will be, indicating how inferior the branch is. When the number of requested slots exceeds the number of available slots that branch can provide, the branch metric becomes infinity, indicating that branch cannot satisfy the request and will be discarded. The function of the branch metric is as follows:

$$B=\begin{cases} f(r,a,w), & r\le a \\ +\infty, & r>a \end{cases}$$

In this work, we assume simplified branch metric reflecting resource allocation status i.e. the branch metric can only have two possible values, either 1 or infinity according to:

$$B=\begin{cases} 1, & r\le a \\ +\infty, & r>a \end{cases}$$

Therefore, the accumulation operation of branch metric in path metric can be omitted. As long as the branch metric is infinity, that branch will be discarded; if the branch metric is 1, that branch might be selected. In our system, every slot at the initial stage has such a representation of trellis graph, i.e. if the slot table size is S, there are S representations of trellis graph. Thus every slot from the initial stage has its own trellis graph and can search its own path in parallel with the others. Consequently, during a search, at each stage we only need to know the branch state at a specific slot. Since the branch at each slot only has two possible status, either free or unavailable (i.e. already allocated), the branch metric of each slot can be simplified as:

$$B=\begin{cases} 1, & \text{branch is free} \\ +\infty, & \text{branch is unavailable} \end{cases}$$

5. *Path metric*: The path metric is the minimal accumulated branch metric over the shortest path from the initial state to the current state. For each state, the incoming branch that produces minimal path metric is selected as the survivor path. The branch metric for a transition from state i to state j at stage n is $B_{i,j,n}$. $P_{j,n}$ defines the path metric for state j at stage n, and S_j is the set of states that have transitions to state j, then:

   $$P_{j,n}=min_{i\in S_j}[P_{i,n-1},B_{i,j,n}]$$

   The goal of the shortest path problem is to find the path between source and destination node with the minimal path metric. At the last stage, the survivor path with the minimal path metric is the desired path. The principle of branch and path metric computation is illustrated in figure 58.

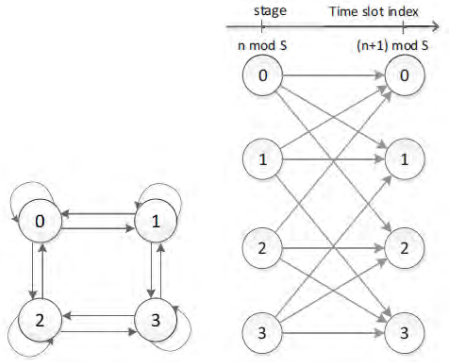


Figure 57: Network graph structure (left) and the associated trellis graph of path search algorithm (right).

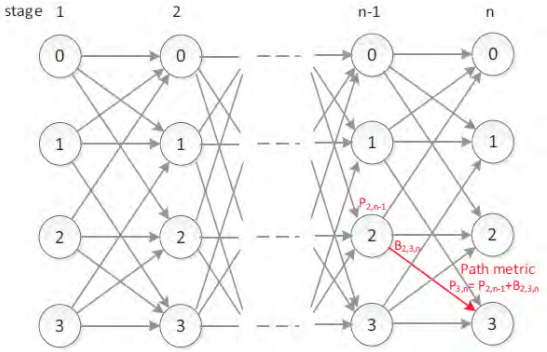


Figure 58: Trellis graph comprising n stages including branch and path metrics for connection optimization.

9.3.2 Trellis Search Structures

In this section, we propose three different trellis structures: unfolded trellis, folded trellis and bidirectional trellis. In general, the shortest path search in trellis by the NoCM comprises two steps:

- Forward search i.e. traverse the NoC from source node to find the best free path to the destination node.
- Backtracking i.e. sort out the saved survivor path from destination to backtrack the shortest path and collect the associated path and slot allocation parameters.

Unfolded Trellis Search: Since the NoC topology and size is known at design time, the respective trellis graph can be constructed as e.g. for 2x2 NoC illustrated in figure 59a the associated trellis graph is shown in figure 59b. Assume node 0 is the source (Src) and node 3 is destination (Des). The search signals propagate from Src at the first stage through the trellis to try to activate its connected neighbours at next stage. Src activates its connected neighbours node 1 and node 2 at the second stage, and then node 1 and node 2 try to activate their connected neighbours at the third stage. Assume the edge *N1* →


N3 at second stage is already occupied, so node 1 cannot activate node 3. During the forward search, if a node is activated by several nodes at the same stage, only one is remembered as its predecessor. When the Des is active, backtracking is started from Des to backtrack predecessors in order to collect the path information. Node 3 backtracks to its predecessor node 2 at the second stage, and node 2 backtracks to node 0 at the first stage. Now the path from Src to Des is obtained as *N0 → N2 → N3*. Assume the starting slot at Src is *n*, then we can obtain the slot sequence along the path as *{n, (n + 1) mod S, (n + 2) mod S}*.

Folded Trellis Search: The proposed unfolded trellis path search algorithm exhibits regular structure and, hence, can be efficiently mapped on to a folded architecture. In such case, the hardware resources are reused by all partitions of folded algorithm. The folded architecture requires additional output registers in order to hold the values of intermediate results to be used as input values in the next iteration. The folded path search algorithm of example in figure 59b is illustrated in figure 59c. There is a register at each node to store which predecessor activates it, and it only stores the predecessor that activates it first. When a node is active, in next cycle it will forward the search signal to its first stage node, and does the propagation search again. Note now the total search costs multiple cycles, i.e. one cycle per iteration. The search can be stopped in two cases: i) either the target node has been activated with sufficient bandwidth or ii) there are no new nodes being activated during the search any more. Hence, in this manner livelock is avoided. In the example shown in figure 59c the search signals start from Src and activate node 2. In the next cycle, node 2 continues to activate node 3. The backtrack (shown in red) starts from destination node 3 and sorts out nodes in sequence N3 → N2 → N0. Therefore, the path from Src to Des is acquired as N0 → N2 → N3.

Bidirectional Trellis Search: The path search presented in the previous sections is started at one side, i.e. from source node to target node. However, it is possible to start the path search from both the source and destination sides simultaneously. If the two searches meet in the middle, then the path search has been successful. The bidirectional path search algorithm of example in figure 59b is illustrated in figure 59d. The search from Src activates node 2, and the search from Des also activates node 2. At the middle stage, the search signals from Src and Des meet at node 2, which means the search is successful. The backtrack starts from node 2 to Src and Des simultaneously. The path from Src to Des is obtained as N0 → N2 → N3. In bidirectional search, the critical path is halved while the area stays almost the same. In the proposed approaches, each slot searches its own path in parallel and can set

up multiple paths. So the communication flow is split over multiple paths, which increases the success rate significantly.

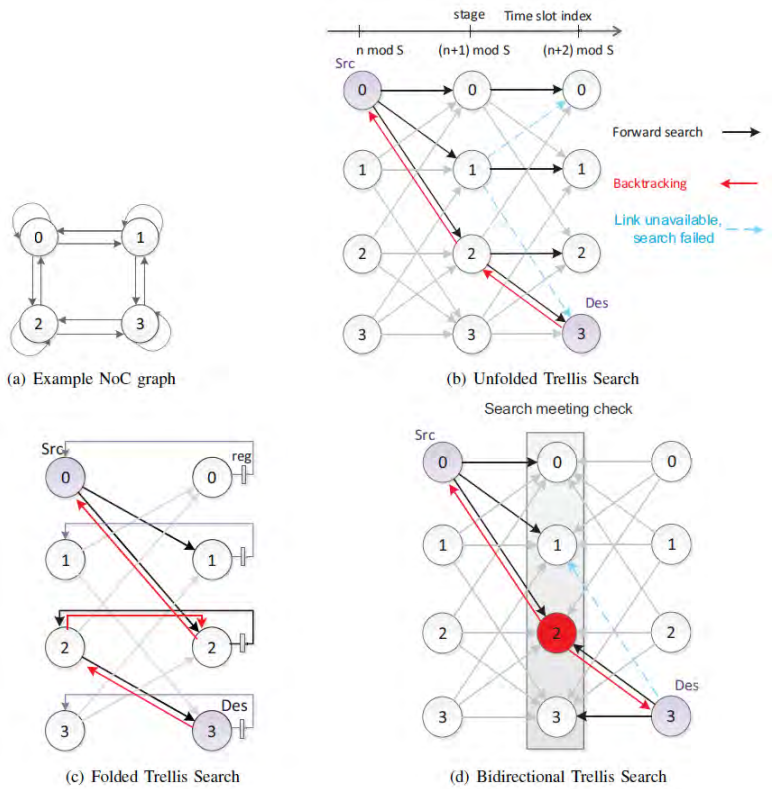


Figure 59: a) Example NoC with 2x2 2D-mesh topology; b) schematic structure of the unfolded trellis; c) schematic structure of the folded trellis; d) schematic structure of the bidirectional trellis.

9.3.3 Trellis Path Search Implementation

This section presents the implementation details of the unfolded trellis and the folded trellis. The Detect-Select-Shift (DSS) Unit is the core module that implements the function of state in the trellis graph, which evaluates the propagated search signals from previous stages and generates bit-vector flags representing slot availability on specific links. The knowledge about the actual link allocation state is stored in the ‘Link State’ register. When a link is allocated at a specific slot, its corresponding state register is set to ‘0’, thereby excluding it from future searches. Correspondingly, state register is set to ‘1’ when the link is released. The architecture of unfolded trellis path search is in figure 60.

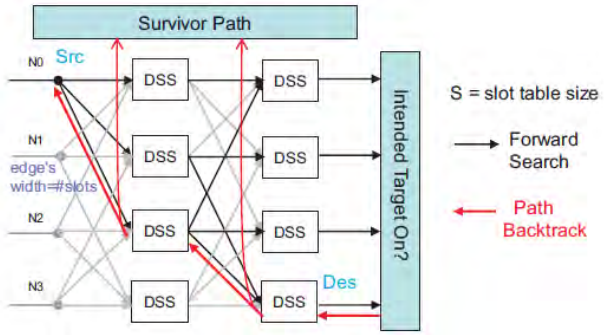


Figure 60: Implementation schematic of the unfolded trellis for the 2x2 mesh NoC.

9.3.4 Performance evaluation

To evaluate the influence of multi-path allocation of TESSA on the allocation success rate, we also realized a single path approach that employs the trellis search algorithm for path search but allocates all required slots (bandwidth) on a single path. This approach is referred to as the single path trellis. The trellis structure of the single path trellis is the same as TESSA except the required bandwidth is allocated on a single path instead of multipath. The allocation speed and success rate of TESSA NoCM are compared to previous centralized and distributed allocation techniques for different NoC sizes with different slot table sizes under synthetic uniform random traffic. These results are explained in the following sections. For evaluation, several performance metrics were considered:

- Success rate denotes the ratio of successful requests that established paths with sufficient bandwidth to the total requests.
- Metrics for comparison with centralized technique: i) background traffic refers to the certain percentage of slots which are already randomly marked as occupied to exclude these slots from path search; ii) allocation time denotes the time that the algorithms need to find out a solution or to determine that the allocation is not possible.
- Metrics for comparison with distributed technique: i) total allocation time denotes the time (nanoseconds) that the algorithms need to find the solution, in addition to the time to send the allocation information to source node. Ii) GS connection Rate denotes the portion of clock cycles in which each master is active sending GS data. Assuming #requested GS connections Q, #flits sent per connection F and #simulation cycles C, the GS connection rate R is computed per master as R=QF/C.


9.3.4.1 Comparison with centralized exhaustive path-search

Folded TESSA is compared to the single path software based exhaustive path-search that runs on Microblaze processor (@288 MHz) [52] in this section.

Comparison of allocation speed: We compare the allocation speed with the exhaustive path-search with 0%, 10% and 20% random background traffic (bk) in a 4x4 mesh network. With different background traffic, the allocation speed of exhaustive path-search is different, but the allocation speed of TESSA is always the same regardless of the background traffic. From figure 61, we can see the allocation time of exhaustive path-search increases linearly in the length of the paths without background traffic (0% bk) and increases exponentially with background traffic (10% and 20% bk). However, the allocation time of TESSA always increases linearly with the path length. The speed of TESSA is about hundreds to thousand times faster than exhaustive path-search. For 6 hops, 12 cycles (12 ns @ 1GHz) are needed for TESSA, *i.e.* 6 cycles needed for forward search and 6 cycles for backtracking, while exhaustive path search requires 8848 ns with 10% background traffic. Hence, TESSA is up to 737 times faster than the exhaustive path-search approach.

Comparison of Success Rate: The requests sent to the NoCM are generated in this way: request to provide an allocation for every feasible source-destination pair combination with certain percentage of background traffic. In our experiments, we produce 1000 samples at each background traffic percentage. We do the simulation for 4x4 meshes with requested slots from 1 to 16 under background traffic from 10% to 50%. The multipath folded TESSA, single path trellis, and software based exhaustive path-search are compared in figure 62. The searching algorithms of single path trellis and single path software approach are similar in that the required bandwidth is allocated over single path, so in the scenarios where the software method’s results are not provided we can imagine it is similar to single path trellis’s. From figure 62 we can see the success rate of folded TESSA can be several to hundred times higher than single path trellis, and it is even higher than exhaustive path-search. Under heavy background traffic with high requested bandwidth, the TESSA becomes far superior to the two single path solutions. E.g. in 4x4 mesh with 16 requested slots, under 20% background traffic, the success rate of folded TESSA is 32X higher than single path trellis and 49X higher than exhaustive path-search, which increases to 103X higher than single path trellis under 50% background traffic.


9.3.4.2 Comparison with distributed parallel probe search

In this section, bidirectional unfolded TESSA is compared to the state of the art distributed parallel probe search [45]. All data points shown in the figures are obtained from simulation over 1 million cycles. The master issues a connection request to the NoCM in a uniform random traffic meeting the requirements of the GS connection rate. The first and last 100,000 simulation cycles were not considered in order to avoid transient effects. The connection lifetime, *i.e.* the number of flits that each connection delivers, is set as 100 flits, 200 flits and 500 flits. During simulation, half of the nodes are assumed as masters and half of the nodes are assumed as slaves. These master nodes are uniformly randomly distributed in the system.

Comparison of allocation speed: In parallel probe search several trials might be needed before success is achieved due to investigation of a single slot at a time. In contrast, in bidirectional TESSA, all slots are searched in parallel, which completes the search in two clock cycles and is independent of the number of slots. Though our design needs additional time to send the allocation information to source node, the path from NoCM to source is found in two cycles by NoCM as a GS path. If the allocation of GS path from NoCM to source node fails, the allocation information will be sent to source as best-effort packets. Because in this simulation setting the GS connection rate is not high (the connection rate is lower than 0.145 in 16x16 mesh and the connection rate is lower than 0.415 in 6x6 mesh), the corresponding allocation success rate is higher than 0.999, so the influence of the allocation failure of GS path from NoCM to source is negligible. For example, in 8x8 mesh with slot table size of 16 at GS connection rate 0.3, the average total allocation time for single slot in probe search is 150 time slots (0.5 x 150 = 75ns). However, in TESSA only 2 clock cycles each are needed for finding the requested GS path and for finding the path from NoCM to source. Since the NoCM is connected to the center router of the NoC, an average of 4 time slots are needed by the NoCM to send the allocation information to source. With a slot table size of 16, there is on average 8 time slots waiting time at the NoCM to get its turn in the TDM scheme. So in total on average 12 time slots (= 4 + 8) in addition to 4 cycles (5.6ns @critical path 1.4ns) are needed, which is 11.6 ns (= 12 x 0.5 + 5.6ns). This is 546% faster than the parallel probe search approach. When n slots are requested, the allocation time for probe search might be increased by n times, but the allocation time for TESSA will be the same as it is independent of the number of requested slots. Hence, when more slots are requested, our solution will present even better results in comparison to the parallel probe search approach. Figure 63 shows the average total allocation time when single slot is

requested for different network sizes. When the GS connection rate is low, the allocation speed of TESSA is similar to parallel probe approach. However, when the connection rate increases, the allocation speed of TESSA becomes much faster than parallel probe. Our approach can provide up to 710% higher speed in 6x6 mesh (@connection rate 0.4), up to 647% higher speed in 8x8 mesh (@connection rate 0.3), and up to 650% higher in 16x16 mesh (@connection rate 0.14). In parallel probe, after the saturation point of the network (connection rate 0.14 in 16x16 mesh and connection rate 0.41 in 6x6 mesh), the allocation time will increase dramatically. Consequently, our approach will become far superior to parallel probe.

Comparison of Success Rate: We re-implement the distributed parallel probe search according to [45] for comparison, with retry deadline as 200 cycles. In parallel probe search, when several connections are requested simultaneously, the concurrent searches might block each other. It only searches the minimal path and cannot explore non-minimal paths as in TESSA. Retry before deadline policy is employed, which can stop the search as failure before all slots are investigated even though there might be available paths. According to these factors, our system would have much higher success rate than parallel probe search. Figures 64 and 65 show the comparison results of success rate. From the simulation results we can see that the success rate of our method is higher than that of probe search. For example, in 6x6 NoC with connection rate between 0.6 and 1.0 and with slot table size of 16, our solution offers up to 26% higher success rate (@ connection lifetime of 100 flits). Moreover, our solution offers up to 29% and 24% higher success rate in 8x8 and 16x16 NoC, respectively. From figure 65, we can see that with more slots (16 slots against 8 slots), the success rate becomes higher, which is due to the increased path diversity.

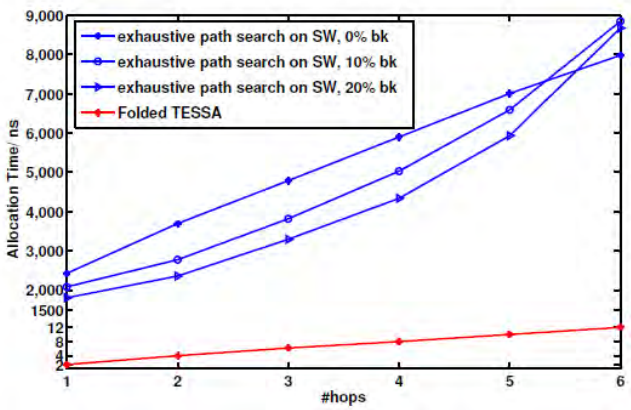


Figure 61: Allocation speed of HW accelerated folded TESSA vs. software-based exhaustive path search on Microblaze-CPU@288MHz for different background traffic and 4x4 NoC with slot table size of 16.

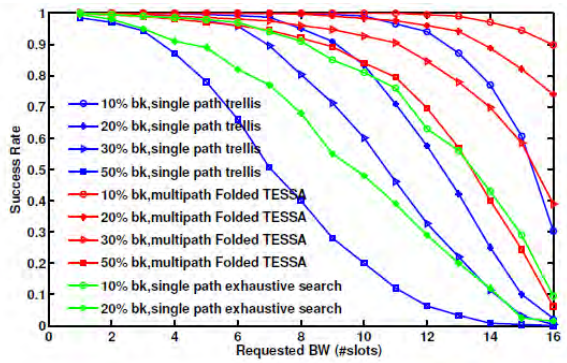


Figure 62: Success Rate compared to single path solutions in 4x4 NoC with different background traffic with Slot Table Size of 16.

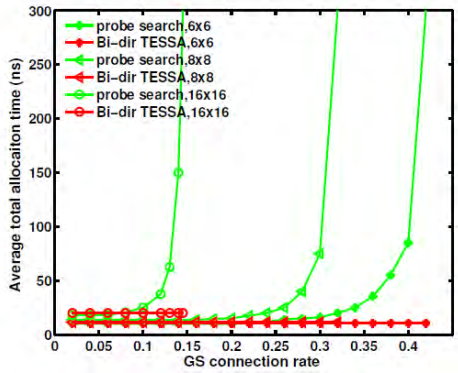


Figure 63: Allocation speed of bidirectional TESSA compared to probe search in different networks with different GS connection Rate with Slot Table Size of 16.

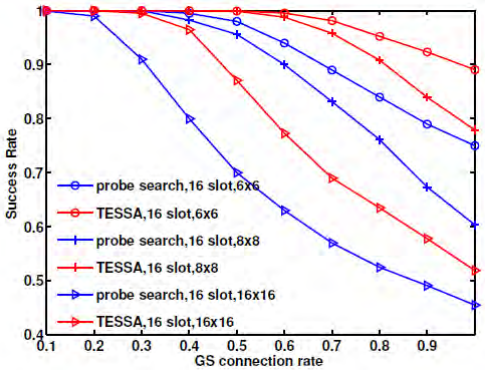


Figure 64: Success Rate of Bidirectional TESSA compared to probe search in different network. Each connection delivers 200 flits.

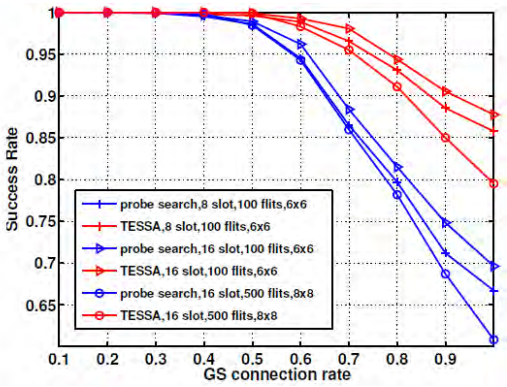


Figure 65: Success Rate compared to probe search in different network with 8 or 16 slots. Each connection delivers 100 or 500 flits.

9.4 Conclusions

In this work we presented a dedicated connection allocator for TDM CS NoCs. In order to meet the requirements of diverse scenarios, three different trellis structures are presented and analysed. In comparison to previous centralized allocations and distributed allocations in terms of allocation speed and success rate, our method achieves highly superior performance. There are four main reasons for our method being superior to other designs: i) The path search problem is solved step by step using dynamic programming approach to reduce computation complexity, as well as to ensure solution optimality (shortest path); ii) multipath slot allocation do search in all directions simultaneously as flooding, which makes success rate and search speed much higher than previous methods; iii) the algorithm structure allows seamless parallelization and hardware acceleration; iv) the algebraic formulation for the system is proposed, which allows complex branch selection criterion to balance the network load and to achieve global optimal results.


10 Conclusion & Future work

This document summarized the improvements brought by the Superfluidity project regarding the problem of resource allocation for network function placement.

We first introduced utility functions, which can give us a flexible objective function to use as an optimization target. Moreover, if you already have statistical knowledge about the performance of your network functions in your NFVI, it might be all that is needed to guide network functions placement in a VM-based environment.

We then proceeded to the extensive characterization of the relation between module placement within hardware components and performance, which led not only to a deeper understanding, but also to great performance improvements, culminating into FastClick, an improved version of the Click Modular Router.

We showed that FastClick was fit-for-purpose, allowing to write easily new network functions while getting excellent performance out of the box, without the need for expert knowledge about the underlying hardware.

To extend the reach of FastClick, it was provided with support for flow-processing. And to further improve its performance, FastClick was extended to automatically eliminate redundant operations across service chains (such as multiple re-classifications), and to support heterogeneous hardware processing (where part of the processing is done on a commodity CPU, and part of the processing on a different device such as an NPU).

We then devised an approach to combine characterization, modeling and optimisation of network function placement using a machine-learning based approach. While this is still a work-in-progress, the required tooling has been implemented, and we began the generation of an open dataset to foster further research into this topic, both in the networking and in the machine learning communities. Preliminary results are very promising, and finishing and exploiting this dataset will be our main focus for further research.

Finally, we also discussed some hypervisor optimizations for VM-based environments, and a novel, real-time and very efficient resource allocation algorithm for resource allocation in the C-RAN.

11 References

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

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

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

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

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

[11] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

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

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

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

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

[16] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the click modular router. In Proceedings of the Asia-Pacific Workshop on Systems, APSYS '12, pages 14:1–14:6, New York, NY, USA, 2012. ACM.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. ACM Trans. Comput. Syst., 18(3):263–297, Aug. 2000.

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

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

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

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

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

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

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

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

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

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

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

[29] H. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. Kaafar, and L. Mathy. SplitBox: Toward Efficient Private Network Function Virtualization. https://arxiv.org/abs/1605.03772. 2016.
[30] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2015.
[31] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In Eurocrypt, 2004.
[32] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehle. Cryptanalysis of the Multilinear Map over the Integers. In Eurocrypt, 2015.
[33] J.-S. Coron, T. Lepoint, and M. Tibouchi. Practical Multilinear Maps over the Integers. In CRYPTO, 2013.
[34] N. A. Jagadeesan, R. Pal, K. Nadikuditi, Y. Huang, E. Shi, and M. Yu. A Secure Computation Framework for SDNs. In HotSDN '14, 2014.
[35] A. R. Khakpour and A. X. Liu. First Step Toward Cloud-Based Firewalling. In SRDS, 2012.
[36] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In NSDI, 2016.
[37] L. Melis, H. J. Asghar, E. De Cristofaro, and M. A. Kaafar. Private Processing of Outsourced Network Functions: Feasibility and Constructions. In ACM Workshop on SDN-NFV Security, 2016.
[38] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In SIGCOMM, 2015.
[39] J. Shi, Y. Zhang, and S. Zhong. Privacy-preserving Network Functionality Outsourcing. http://arxiv.org/abs/1502.00389, 2015.
[40] The Linux Foundation. Packetgen. http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen.
[41] A. Waterland. Stress. http://people.seas.harvard.edu/~apw/stress/.
[42] W. Norcott. IOzone. http://www.iozone.org/.
[43] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Forwarding path architecture for multicore software routers. In ACM PRESTO, 2010.
[44] S. Haas et all, "A Heterogeneous SDR MPSoC in 28 nm CMOS for Low-Latency Wireless Applications," in Proceedings of the Design Automation Conference (DAC'17), Austin, Texas, 2017
[45] Liu S, Jantsch A, Lu Z. Parallel probe based dynamic connection setup in TDM NoCs. Design, Automation & Test in Europe (DATE), 2014.
[46] Goossens K, et al. AEthereal network on chip: concepts, architectures, and implementations. Design & Test of Computers, IEEE, 2005.
[47] Goossens K, Hansson A. The Aethereal network on chip after ten years: Goals, evolution, lessons, and future. DAC, IEEE, 2010.
[48] Lusala A K, Legat J D. Combining sdm-based circuit switching with packet switching in a NoC for real-time applications. ISCAS, 2011.
[49] Lusala A K, Legat J D. Combining SDM-Based circuit switching with packet switching in a router for on-chip networks. International Journal of Reconfigurable Computing, 2012.
[50] Liu S, et al. A fair and maximal allocator for single-cycle on-chip homogeneous resource allocation. VLSI, IEEE Transactions on, 2014.
[51] Chen Y, Matus E, Fettweis G P. Trellis-search based Dynamic Multi-Path Connection Allocation for TDM-NoCs. GLSVLSI. ACM, 2016.
[52] Stefan R, Nejad A B, Goossens K. Online allocation for contention-freerouting NoCs. Interconnection Network Architecture: On-Chip, Multi-Chip Workshop. ACM, 2012.
[53] Lu Z, Jantsch A. TDM virtual-circuit configuration for network-on-chip. VLSI, IEEE Transactions on, 2008.
[54] Schoeberl M, et al. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. NoCS, IEEE, 2012.
[55] Kasapaki E, Spars J. Argo: A time-elastic time-division-multiplexed NOC using asynchronous routers. ASYNC, IEEE, 2014.

[56] Winter M, Fettweis G P. A network-on-chip channel allocator for runtime task scheduling in multi-processor system-on-chips. DSD'08.
[57] J. Xu and J. A. B. Fortes. Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments. In Proceedings of IEEE/ACM International Conference on Green Computing and Communications (GreenCom) & International Conference on Cyber, Physical and Social Computing (CPSCom), 2010.
[58] H. Jin, D. Pan, J. Xu and N. Pissinou. Efficient VM placement with multiple deterministic and stochastic resources in data centers. In Proceedings of the IEEE Global Communications Conference (GLOBECOM), 2012.
[59] X. Li, A. Ventresque, J. Murphy and J. Thorburn. A Fair Comparison of VM Placement Heuristics and a More Effective Solution. In Proceedings of the 13th IEEE International Symposium on Parallel and Distributed Computing (ISPDC), 2014
[60] John Wilkes. Utility Functions, prices and negotiations. 2008.
[61] M. Macias and J. Guitart. Using resource-level information into nonadditive negotiation models for cloud Market environments. In Proceedings of IEEE Network Operations and Management Symposium (NOMS), 2010.
[62] A. Tumanov, T. Zhu, M. A. Kozuch, M. Harchol-Balter and G. R. Ganger. TetriSched: Space-time scheduling for heterogeneous datacenters. Parallel Data Laboratory, Carnegie Mellon University, 2013.
[63] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In In Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys, 2012.
[64] D. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. IEEE Journal on Selected Areas in Communications, vol. 24, pp. 1439-1451, 2006.
[65] J.-W. Lee, M. Chiang and A. Calderbank. Price-based distributed algorithms for rate-reliability tradeoff in network utility maximization. IEEE Journal on Selected Areas in Communications, vol. 24, no. 5, pp. 962-976, 2006.
[66] T. Metsch, O. Ibidunmoye, V. Bayon-Molino, J. Butler, F. Hernández-Rodriguez, and E. Elmroth. Apex Lake: A Framework for Enabling Smart Orchestration. In Proceedings of the Industrial Track of the 16th International Middleware Conference (Middleware Industry '15). ACM, 2015. http://dx.doi.org/10.1145/2830013.2830016.
[67] Ios, Pearson Correlation and Pearson Squared, Available: http://www.improvedoutcomes.com/docs/WebSiteDocs/Clustering/Clustering_Parameters/Pearson_Correlation_and_Pearson_Squared_Distance_Metric.htm
[68] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter and G. Shi. Design and implementation of a consolidated middlebox architecture. In proceedings of the 9th UNENIX conference on Networked Systems Design and Implementation. 2012.
[69] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. ACM SIGCOMM Computer Communication Review 42, 4. 2012.
[70] Z. Wang, Z. Qian, Q. Xu, Z. Mao and M. Zhang. An untold story of middleboxes in cellular networks. In proceedings of the ACM SIGCOMM 2011 conference. 2011.
[71] Cisco. Snort: Network Intrusion Detection & Prevention System. http://www.snort.org/.
[72] HAProxy: The reliable, high performance TCP/HTTP load balancer. http://www.haproxy.org/.
[73] NGINX Inc. NGINX: High performance load balancer, web server & reverse proxy. https://www.nginx.com/.
[74] M. V. Bernal, I. Cerrato, F. Risso and D. Verbeiren. Transparent optimization of inter-virtual network function communication in open vSwitch. In 5th IEEE Internation Conference on Cloud Networking (Cloudnet). 2016.
[75] L. Rizzo and G. Lettieri. VALE, a switched Ethernet for virtual machines. In proceedings of the 8th internation conference on Emerging Networking Experiments and Technologies. CoNext'12. 2012.
[76] NPF: Network Performance Framework. https://github.com/tbarbette/npf.
[77] H. Mao, M. Alizadeh, I. Menache and S. Kandula. Resource management with deep reinforcement learning. In Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets). 2016.

[78] Gradient Boosting Regression Trees. https://en.wikipedia.org/wiki/Gradient_boosting.