SUPERFLUIDITY

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

Research and Innovation Action GA 671566

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

Deliverable Type:	Report
Dissemination Level:	PU
Contractual Date of Delivery to the EU:	M20 (1/03/2017)
Actual Date of Delivery to the EU:	M20 (10/03/2017)
Workpackage Contributing to the Deliverable:	WP5

Editor(s):	ULG
Author(s):	**ULG (Cyril Soldani, Tom Barbette, Laurent Mathy), INTEL (Radhika Loomba, Michael Mcgrath)**
Internal Reviewer(s)	Bessem Sayadi (NOK-FR)

Abstract:	This deliverable describe progress on the task 5.1 related to the optimal placement

	of network functions in the underlying infrastructure.
Keyword List:	Automated resource allocation, Optimisation, Performance

INDEX

GLOSSARY....6
1 INTRODUCTION....8
1.1 Deliverable Rationale....8
1.2 Quality Review....8
1.3 Executive summary....8
1.3.1 Deliverable description....8
1.3.2 Summary of results....9
2 IMPROVEMENTS TO FASTCLICK....10
2.1 Heterogeneous packet processing....11
2.2 MiddleClick: Click-based flow-processing middleboxes....13
2.2.1 Common problems with traditional middleboxes....13
2.2.2 Introducing MiddleClick....15
3 NETWORK PERFORMANCE FRAMEWORK....17
3.1 Comparison mode....18
3.2 Regression mode....19
3.3 Statistical analysis....20
3.3.1 Best value....21
3.3.2 Regression tree....21
3.4 Software definitions....22
3.5 Conclusion....23
4 UTILITY FUNCTIONS FOR OPTIMAL RESOURCE ALLOCATION....23
4.1 Introduction....23
4.2 Resource Representations and Mapping....24
4.2.1 NFVI Landscape....24
4.2.2 Resource Request....26
4.2.3 Service Level Agreement....26
4.2.4 Mapping Assumptions....26
4.3 Deployment using Multi Attribute Utility Theory....27
4.4 Utility Function Definitions....28
4.4.1 Performance Viability....28
4.4.2 Economic Viability....29
4.4.3 Service Distribution....30
4.5 Composing a Deployment Utility Function....30
4.6 Rebalancing....31

4.7 Future Plan....31
4.8 Conclusions....31
5 MACHINE-LEARNING BASED OPTIMISATION....32
5.1 Joint modelling and optimisation for function allocation....32
5.2 Challenges....34
5.3 Action plan....35
5.3.1 Click-based environments....35
5.3.2 VM-based or container-based environments....35
6 CONCLUSION & FUTURE WORK....36
7 REFERENCES....37


List of Figures

Figure 1: Comparison of router throughput for FastClick and best state-of-the-art Click I/O implementations (using 4 cores except for in-kernel Click).....11
Figure 2: Example of configuration split for heterogeneous packet processing.13
Figure 3: Different ways to build a middlebox service chain. On most links, there is no cooperation to avoid redundant operations.....14
Figure 4: Data downloaded through a load-balancer using 128 concurrent connections. This is equivalent to 130 000 requests/s with 8-kB files.....17
Figure 5: Evaluating the impact of zero-copy option of iperf.....18
Figure 6: Comparing iperf and netperf.....19
Figure 7: Click-based performance regression test. By default, the regression tool will test the last 10 git commits for git-managed software.....20
Figure 8: NPF-generated performance regression tree.....22
Figure 9: NFVI Landscape Graph.....25
Figure 10: Multi-Attribute Utility Theory workflow.....28
Figure 11: Superfluidity architecture, from D3.1.....32

List of Tables

Table 1: SUPERFLUIDITY Dictionary.....7

Glossary

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
API	Application Programming Interface
ARP	Address Resolution Protocol
ATA	AT (Advanced Technology) Attachment
BF	Bloom Filter
BSD	Berkeley Software Distribution
CLT	Coron Lepoint Tibouchi
CPU	Central Processing Unit
DMA	Direct Memory Access
DPDK	Intel's Data Plane Development Kit
DPI	Deep Packet Inspection
DUT	Device Under Test
FIB	Forward Information Base
FIFO	First-In First-Out
GPL	GNU General Public License
GPU	Graphics Processing Unit
GSO	Generic Segmentation Offload
I/O	Input / Output
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IOV	I/O Virtualization
IP	Internet Protocol
IPS	Intrusion Prevention System
IRQ	Interrupt ReQuest
KPI	Key Performance Indicator
MAUT	Multi-Attribute Utility Theory
MB	MiddleBox or MegaByte
MQ	Multi-Queue
NAT	Network Address Translation


NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NIC	Network Interface Controller
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
RAM	Random Access Memory
RAN	Radio Access Network
REE	RFB Execution Environment
RFB	Reusable Function Block
RSS	Receiver-Side Scaling
RX	Receive
SDN	Software Defined Network
SFP	Small Form-factor Pluggable
SLA	Service Level Agreement
SR-IOV	Single-Root IOV
TCP	Transmission Control Protocol
TSO	TCP Segmentation Offloading
TTL	Time-To-Live field of IP header
TX	Transmit
UDP	User Datagram Protocol
vCPU	Virtual CPU
VM	Virtual Machine
ZC	Zero Copy

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

1.1 Deliverable Rationale

This internal deliverable is meant to share progress on the 5.1 task (function allocation optimisation) to facilitate collaboration between the involved partners, and inform the project as a whole of the advancement of the task.

1.2 Quality Review

Review Team member responsible of the deliverable: _____

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE

1.3 Executive summary

1.3.1 Deliverable description

This deliverable is an intermediate advancement report about the work-in-progress on task 5.1, *Function allocation algorithms implementation and evaluation*. It also discusses current plans to achieve the task goals by the end of the project.

This task tries to leverage the block and functional abstractions developed in Tasks 4.2 and 4.3 to match a set of network services (decomposed into functional abstractions) to the underlying, available block hardware abstractions. This task will identify SLA (Service Level Agreement) metrics (*e.g.* throughput, delay, power consumption, *etc.*), and manage their mapping onto platform resources. The problem solved by this task is thus the optimization problem of minimizing the platform resource usage while meeting individual SLAs, in order to automatically derive the best performance out of a platform. The work will include implementing algorithms that solve the allocation


problem and evaluating for efficiency (*e.g.* in terms of goodput, delay, packet loss and number of services) when asked to install different sets of network services. The algorithms will further include dynamic re-evaluation of allocations in real-time, in order to adapt to changing traffic patterns (*e.g.* to pin virtual machines to different CPU cores in order to better cope with a traffic spike).

For instance, we will focus on dynamic allocation and scheduling techniques for data-flow task graphs representing protocol stack applications. For this purpose, we will make use of hardware abstractions developed in WP4. Decomposition of network processing functions in blocks allows exploiting inherent parallelism and heterogeneity of applications for workload distribution to processor/thread pools, resulting in accelerated execution of the applications. Currently, static methods are usually preferred for task mapping, however, these suffers from inefficient use of resources resulting in system over-provisioning.

1.3.2 Summary of results

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

Previously:

- We first sought to characterize the performance of network traffic processing pipelines within stand-alone hardware processing unit. To this end, we have analysed 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 gave insights on how to allocate networking functions onto a stand-alone machine, and led to a low-level component allocation framework called FastClick (section 2 of deliverable I5.1).
- We have also studied FastClick's fitness for purpose through the development of a novel network cloud function called SplitBox, a privacy preserving filtering function for outsourced firewalls. This application showed that FastClick heuristic resource allocation algorithms on a stand-


alone machine make a good use of available resources, providing good performance for a demanding application, while requiring no advanced knowledge of the hardware platform from the network function implementer (section 3 of deliverable I5.1).

- We have also investigated the characterization of a virtualized NFV infrastructure under different workloads (section 4 of I5.1).
- We also investigated how our performance improvements on a bare-metal machine could translate into a virtualized setting, using the MicroVisor platform (section 5 of deliverable I5.1).

Since then, several improvements have been made to the FastClick platform, which we describe in section 2 of this document.

To help us investigate the impact of various parameters on performance in the context of middleboxes, we developed a Network Performance Framework (NPF) to assist both in running experiments, and analyse their results using statistical analysis and machine learning. This tool is described in section 3.

Following previous work done in I5.1, but also in tasks 4.1 and 4.2, we propose a framework based on **utility functions** to bridge telemetry-collected data about the NFV Infrastructure (NFVI), SLAs agreed between the owner of the NFVI and its customers, and the deployment of corresponding service chains. This work is developed in section 4.

Utility functions are expected to provide a good way to evaluate the mapping between application requests and the underlying infrastructure. However, the configuration space of the NFVI (*i.e.* the possible number of configurations) grows exponentially with the number of processing nodes, and the number of possible parameter values for each node. It is thus not possible to evaluate all possible deployments to rank them according to their expected utility. We need to intelligently reduce the exploration to only a small part of the configuration space. Moreover, the behaviour of the system being subject to many non-linear effects, it is hard to optimise using traditional optimisation techniques. We propose to attack this problem using techniques based on **machine-learning**. Our approach is described in section 5.

2 Improvements to FastClick

FastClick [25] was presented in details in previous deliverable I5.1. It is an extension of the Click Modular Router [12], which optimises and simplifies packet processing on commodity hardware.

After thorough characterisation of packet I/O frameworks and packet processing execution models, we have been able to achieve:

- improved packet I/O **performance** through well-engineered userlevel I/O (using netmap or DPDK);
- reduced processing overhead (full-push mode, batching, *etc.*);
- *some* **automatic resource allocation** (cores, NIC queues, *etc.*), which simplifies configuration. This automatic resource allocation uses heuristics. While it shows good performance for most applications, it is not always optimal and can be overridden by the user.

Figure 1 illustrates the performance improvement on a router test case. FastClick was also tested in a cloud-based network application: SplitBox [24], a firewall outsourced to the cloud while keeping the security policy private from the cloud operators, with good performance and relative ease of configuration.

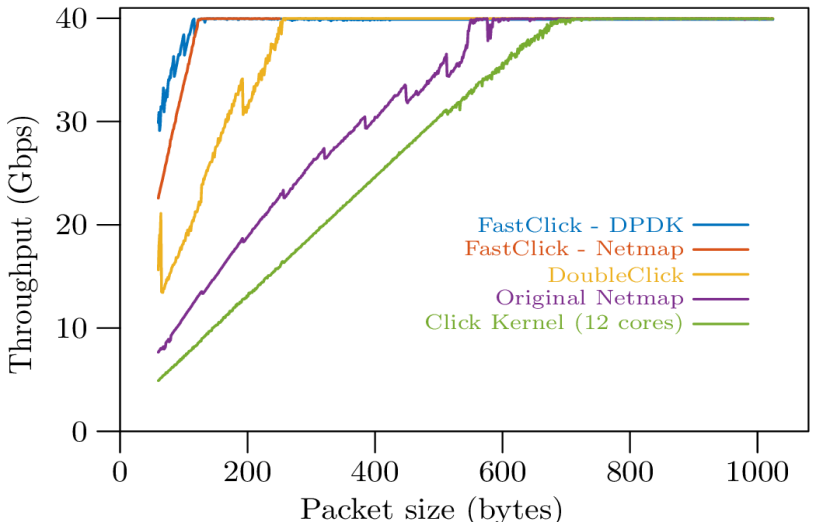

Figure 1: Comparison of router throughput for FastClick and best state-of-the-art Click I/O implementations (using 4 cores except for in-kernel Click).

Since deliverable I5.1, we contributed various improvements to FastClick, but the most important ones are the following: the implementation of a mechanism to transfer packets between multiple Click instances running on different hardware components in the same machine, and the implementation of an extension of FastClick, called **MiddleClick**, that adds support for flow-processing.

2.1 Heterogeneous packet 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 next 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.

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 ULg 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 2 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.

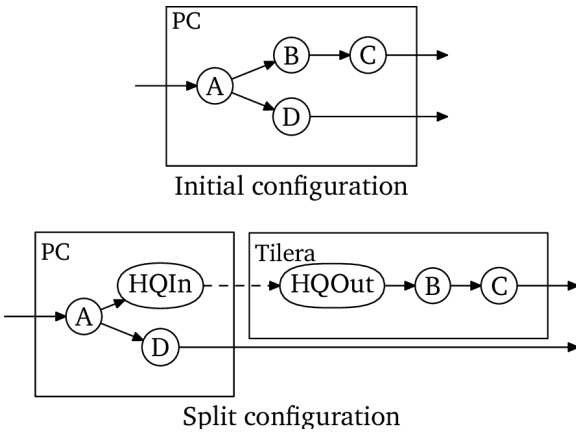

Figure 2: Example of configuration split for heterogeneous packet processing.

2.2 MiddleClick: Click-based flow-processing middleboxes

2.2.1 Common problems with traditional middleboxes

According to [49][50], there are roughly as many middleboxes as routers in enterprise networks. A myriad of middleboxes is also deployed in increasingly important mobile networks [51], 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.

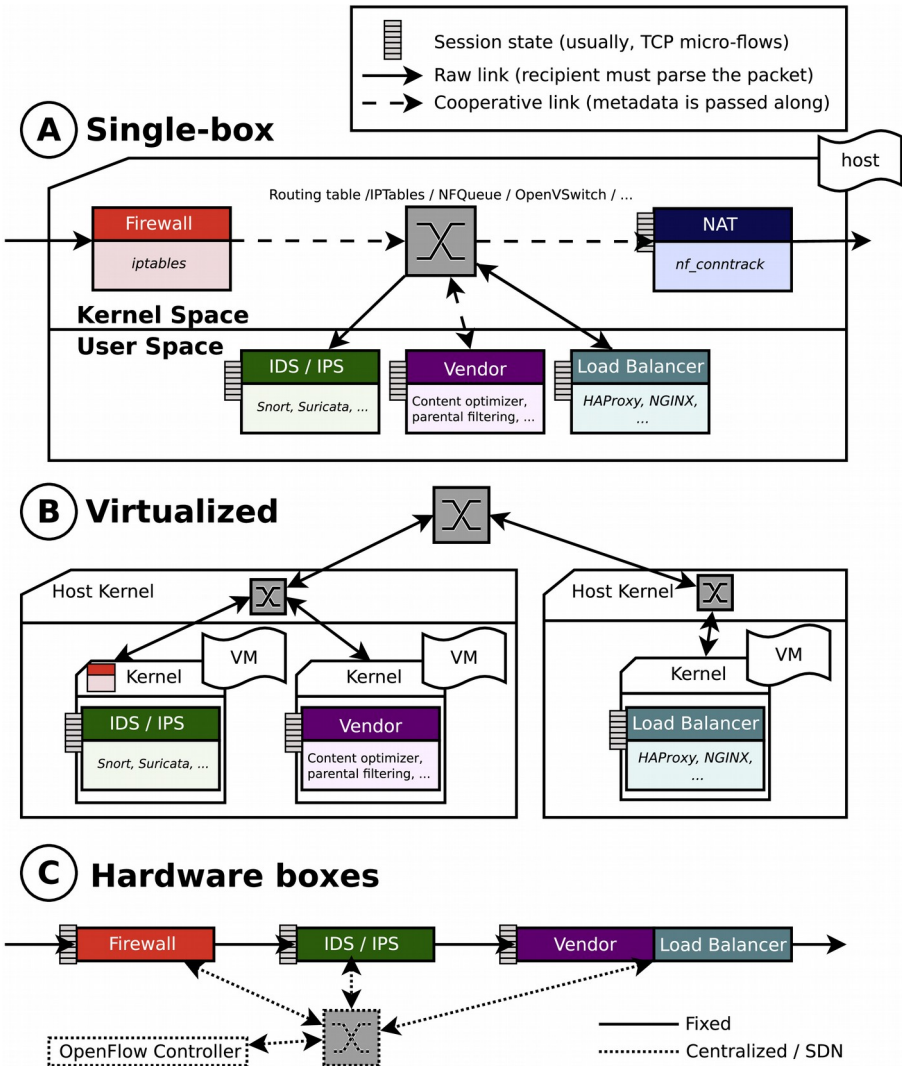

Figure 3: Different ways to build a middlebox service chain. On most links, there is no cooperation to avoid redundant operations.

Three typical implementations of a service chain are shown on figure 3. 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 [52] for IDS, NetFilter/IPTables/NFT for the firewall, HAProxy [53] or NGINX [54] 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 [55][56].

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.

2.2.2 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 allows to handle a very large amount of concurrent sessions by providing support for 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 avoid 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 then 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 4 shows a performance comparison between HAProxy, when used as a TCP load-balancing reverse proxy, and the same functionality implemented using MiddleClick.

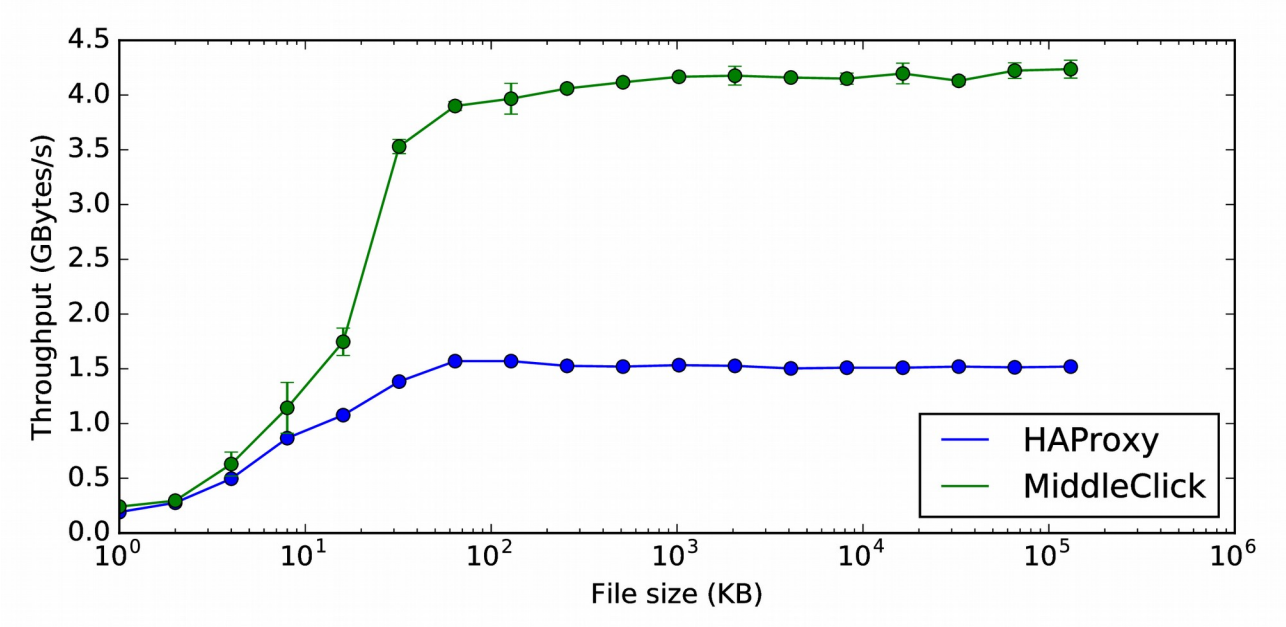

Figure 4: Data downloaded through a load-balancer using 128 concurrent connections. This is equivalent to 130 000 requests/s with 8-kB files.

3 Network Performance Framework

Network Performance Framework (NPF) [58] 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.

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.

3.1 Comparison mode

Figure 1 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 LOCALHOST -P $PARALLEL \
$ZEROCOPY | TAIL -N 3 | GREP -IOE "[0-
9.]+ KBITS"
```

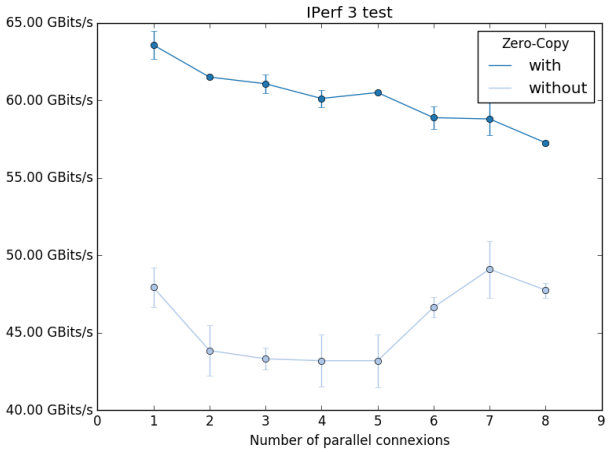

Figure 5: Evaluating the impact of zero-copy option of iperf.

%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 2 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 LOCALHOST -P $PARALLEL -Z | TA
-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"
```

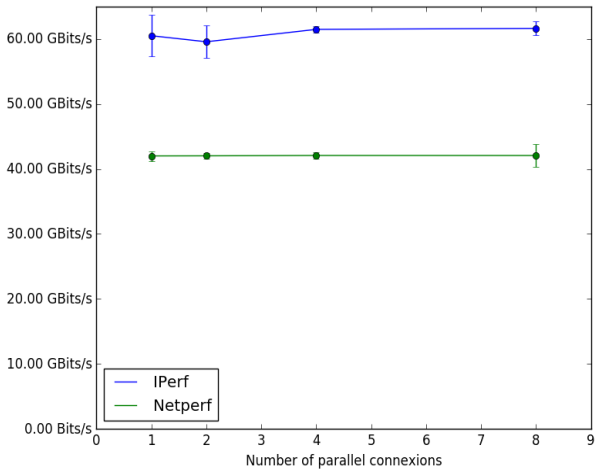

Figure 6: Comparing iperf and netperf.

3.2 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 3.4). 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))")
```

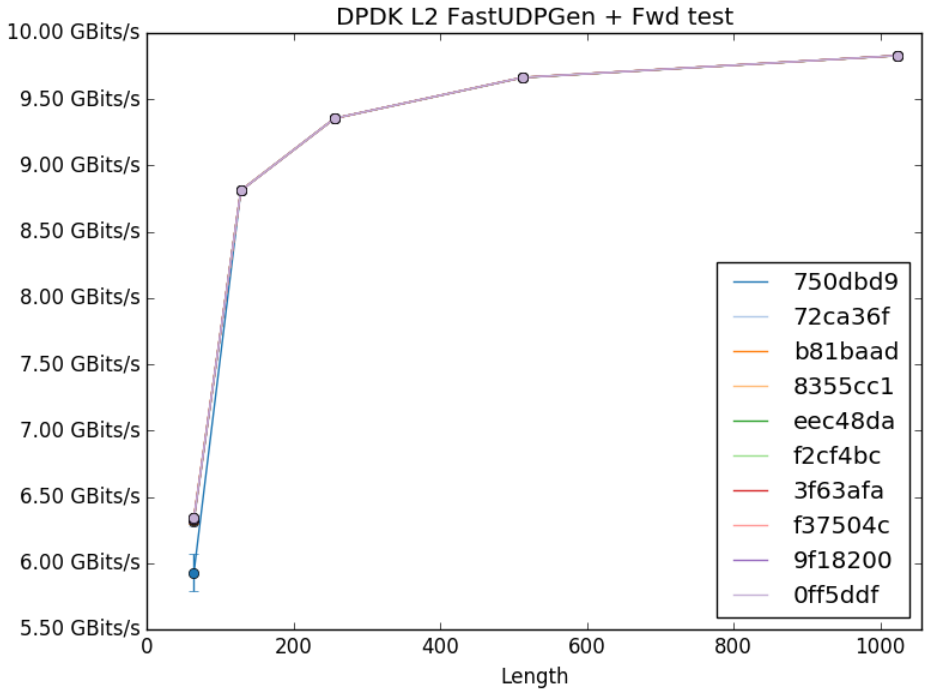

Figure 7: Click-based performance regression test. By default, the regression tool will test the last 10 git commits for git-managed software.

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

3.3 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 replay 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.

3.3.1 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
```

3.3.2 Regression tree

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

```
Feature importances :
 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.

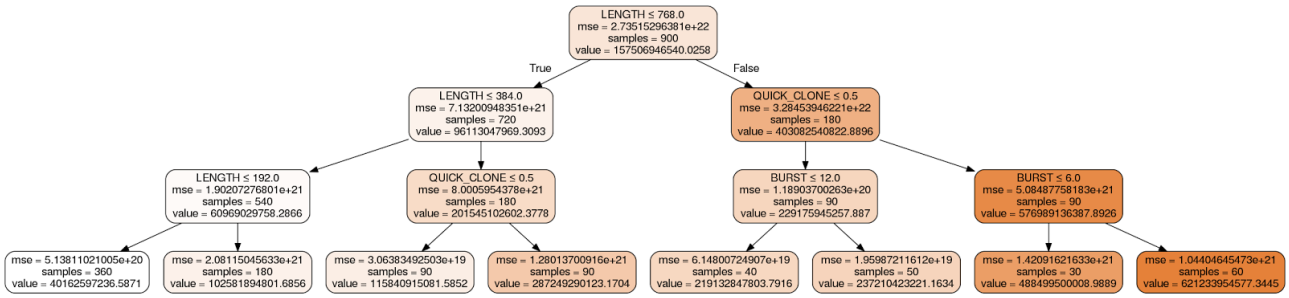

Figure 8: NPF-generated performance regression tree.

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.

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

3.5 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.

In the future, we intend to extend 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. Such mechanisms are also likely to be useful for resource allocation, as we will explain in section 5.

4 Utility functions for optimal resource allocation

4.1 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 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 [39], stochastic bin-packing methods [40] and multiple heuristics [41]. 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 *vs* the service customer.


This work investigates 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 [42] and Macias *et al.* [43], and been used by several researchers for specifying the customer's preferences, *e.g.* for scheduling in Tetrisched [44], where the authors focus on predicted runtimes and job-preferences, and Jockey [45], with dynamic estimation for maximising job utility. Network Utility Maximisation [46] is another area which looks at improving throughput of a network by maximizing utility with trade-offs between rate and reliability [47] 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 therefore allows us to minimise 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 visualization of possible solutions. This has the added benefit of providing intelligence to the 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.

4.2 Resource Representations and Mapping

4.2.1 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 [48]. Each resource node within a layer can be divided into three distinct functional elements namely compute, network and storage as shown in figure 9.

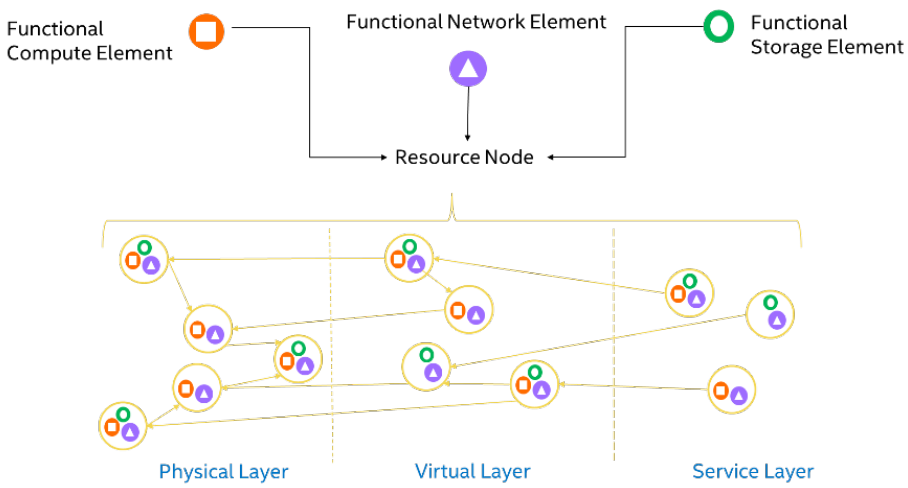

Figure 9: NFVI Landscape Graph.

Each functional element in the resource node $x \in X$ is linked to two capacity values, namely the resource and the processing capacity. The resource capacity is used to define the **platform server components** available. Mathematically, the compute resource capacity is r_x^c , which can be defined in terms of the number of cores, the network resource capacity is r_x^n , which can be defined in terms of the available bandwidth and the storage resource capacity is r_x^s, which can be defined by the storage size.

The processing capacity quantifies the **utilization and saturation** values which indicate **goodput** (application-level throughput), **packet loss** *etc.* depending upon the category of the functional element. Mathematically, these are represented as p_x^c for compute, p_x^n for network and p_x^s for storage.

Additionally, several parameters are defined for each node, with subscripts c, n, s to denote the functional elements. These include utilization $U_x^c \in [0,1]$, $U_x^n \in [0,1]$, $U_x^s \in [0,1]$ and saturation $S_x^c \in [0,1]$, $S_x^n \in [0,1]$, $S_x^s \in [0,1]$ which are normalized for time δt . Other parameters include R_x^c, R_x^n, R_x^s denoting the rate of provisioning resource capacity per hour, P_x^c, P_x^n, P_x^s denoting the rate of provisioning processing capacity per hour per provider, 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 $e \in E$ has associated with it maximum bandwidth B_e, with rate R_e^b for a unit of bandwidth and geographical distance covered $len(e)$ which remain constant. The latency of the link l_e and throughput defined using available bandwidth τ_e are normalized for time δt.

4.2.2 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 resources of compute, network and storage capacity are indicated by $\varphi_z^c, \varphi_z^n, \varphi_z^s$ and the requested processing compute, network and storage capacities are indicated by $\phi_z^c, \phi_z^n, \phi_z^s$ respectively 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.

4.2.3 Service Level Agreement

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

- Total running time for the request, represented by T_r hours
- Rate for each template $SLA \in SLAs$, defined as R_{SLA}
- Unit cost of SLA Violations represented as $j \in J$,
- The list of failure-tolerant implementations $h \in H$.
- KPI's to be fulfilled by the deployment, *e.g.* latency and throughput

4.2.4 Mapping Assumptions

The request nodes $z \in Z$ are mapped on 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:

1. τ_e is flow on this edge is equal to b_f

2. $l_f = l_e$

4.3 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.

1. Performance Viability
2. Economic Viability
3. Service Distribution

The methodology supports ranking of all deployment solutions from the least to most desirable. Figure 10 shows the process flow which is currently in development (see I4.1. for details on KPI Mapping).

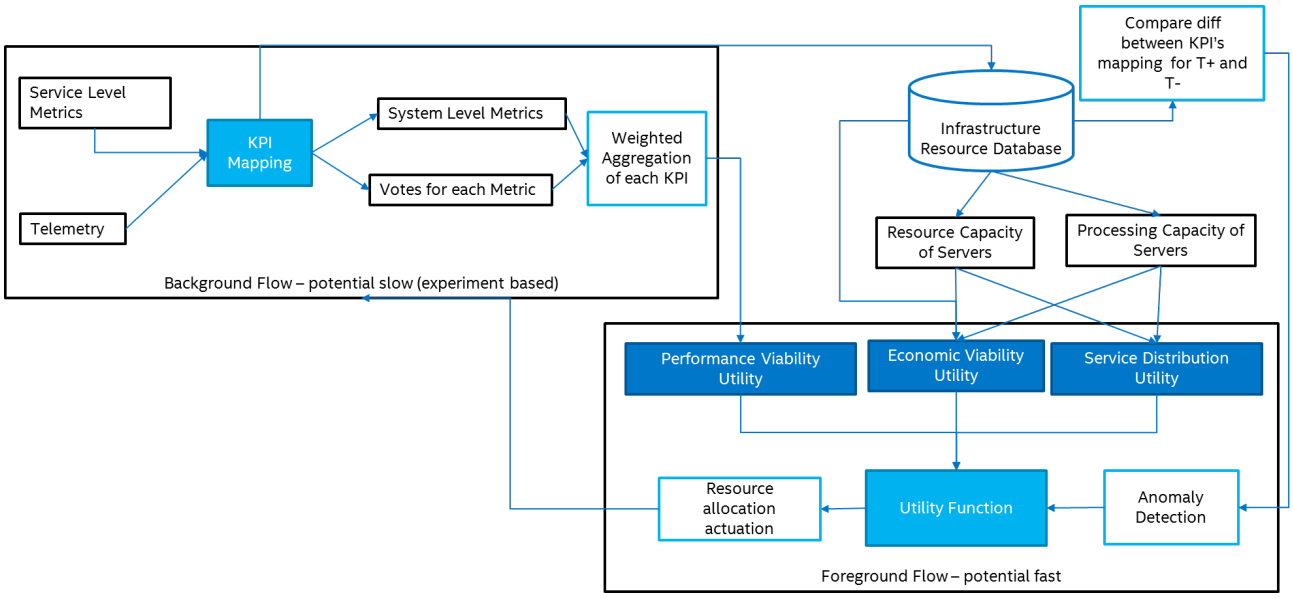

Figure 10: Multi-Attribute Utility Theory workflow.

4.4 Utility Function Definitions

4.4.1 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 I4.1) 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 (see I4.1), 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 $U()$ 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: $U(P_Z) = \sum_i v_i * i$.

4.4.2 Economic Viability

This utility is used to investigate the fulfilment 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, taking into account the resource and processing capacity of the resource nodes. 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:

- Resource Revenue = $T_r * \sum_{z \to y \in Y} \varphi_z^c * R_y^c + \varphi_z^n * R_y^n + \varphi_z^s * R_y^s$
- Processing Revenue = $T_r * \sum_{z \to y \in Y} \phi_z^c * P_y^c + \phi_z^n * P_y^n + \phi_z^s * P_y^s$
- 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) = $T_r * \sum_{j \in J} j * \bar{j}$.

Then depicting 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 $U()$ for the economic viability for deployment Z is represented by $U(E_Z)$ and formulated as below:

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

4.4.3 Service Distribution

This utility functions is used to quantify how the resources are distributed over the different servers/nodes. Two rules are used for quantifying a better deployment, namely consolidation of the service within a server (inversely proportional to the leftover capacity after deployment), and minimal fragmentation (inversely proportional to the number of servers which have a capacity > 0 after deployment). Thus a higher utility indicates that the **minimal number of resources** are used for the deployment, and more resource as well as processing capacity are available after deployment of service Z .

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

$$U(S_Z) = \theta_1 * U(r_Z) + \theta_2 * U(p_Z)$$

Here θ_1 and θ_2 denote weights for the utility determining resource capacity $U(r_Z)$ and the utility determining the processing capacity $U(p_Z)$ respectively. These are defined as below:

$$U(r_Z) = \frac{|Y|}{\sum_y (r_y^c - \varphi_z^c) + (r_y^n - \varphi_z^n) + (r_y^s - \varphi_z^s)} * \frac{1}{\sum_{x: r_x > 0} 1}$$

$$U(p_Z) = \frac{|Y|}{\sum_y (p_y^c - \phi_z^c) + (p_y^n - \phi_z^n) + (p_y^s - \phi_z^s)} * \frac{1}{\sum_{x: r_x > 0} 1}$$

4.5 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.


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

4.6 Rebalancing

We also propose studying the KPI mapping results within a particular time range to indicate changing configurations and resource infrastructure. If a significant difference is detected, this can indicate an anomaly in the system. In this scenario, the utility functions will be used to initiate rebalancing. This will potentially support dynamic re-evaluation of deployments in real-time. However, this proposal needs to be validated with more experimentation and data analysis.

4.7 Future Plan

In this deliverable we present our initial work on the formulation of a utility based approach to optimally allocate resources and topological configuration of services deployment in order to fulfil KPIs defined in an SLA. The key research activities planned over the next quarters are as follows:

- Execution of an experimental campaign to study the effect of system-level metrics on the KPI (*e.g.* degrades performance or improves performance).
- Execution of an experimental campaign to study variance of KPI mapping results to initiate rebalancing.
- Validation and verification of approach by simulation.
- Validation and verification of approach on a real testbed setup.

4.8 Conclusions

The application of utility functions to study a multi-optimisation deployment problem is currently under investigation. The approach being developed combines sub-utilities of multiple objectives, analysed on a common orchestration platform to consistently rank solutions and thus capture the benefits and shortcomings of each deployment. Initial formulations have been completed and described in this section. These definitions appear to be novel, with limited investigation available in current literature. The final outcomes from this work will be presented in deliverable D5.1.

5 Machine-learning based optimisation

5.1 Joint modelling and optimisation for function allocation

The architecture of Superfluidity is depicted in figure 11. 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.

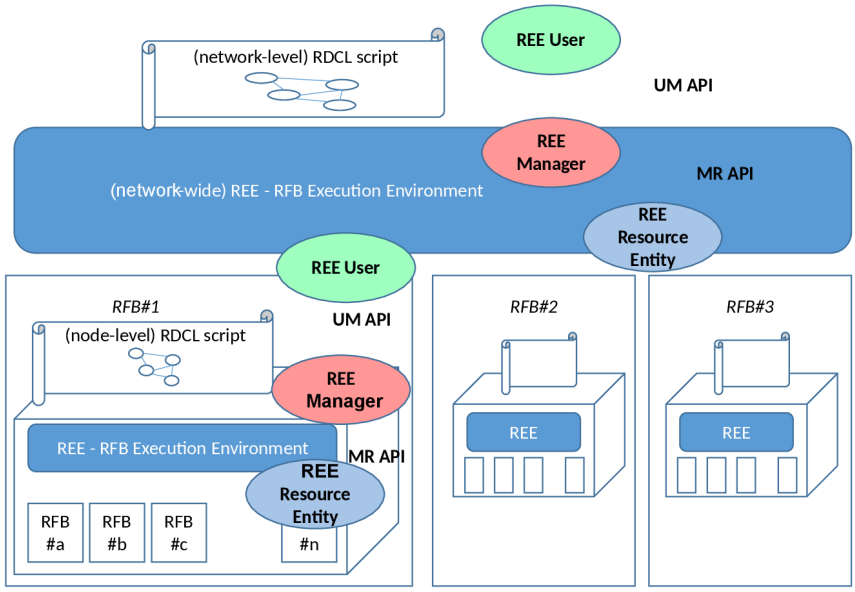

Figure 11: Superfluidity architecture, from D3.1.

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 4 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 3), 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 modelling 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.
- Optional information about the **workload** used for the experiment. As we have shown in I5.1 (section 4), the workload 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).


5.2 Challenges

Although machine learning has already been used to solve some resource allocation problems (*e.g.* [57]), 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 11), 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 yet to be found, and we might not be able to evaluate the performance of subsets of the configuration in isolation.

5.3 Action plan

In the remainder of the project, we will first validate the approach on simple cases and, if it works, expand it to larger, more integrated test cases. We will target two different resource execution environments: Click-based systems and VM-based (or container-based) systems.

5.3.1 Click-based environments

We will first validate our approach by learning actual configurations for single applications (*e.g.* a router), with a single client (*i.e.* just one instance of the application) on a single machine. This will allow us to ensure that the approach is promising (*i.e.* gives better performance than state-of-the-art heuristics), and give us some insights to solve the more challenging problems.

We would then turn to the cases where we have multiple client instances of the same application on one machine. The third step would be to expand the configurations to multiple different network functions (potentially from multiple clients) on one machine. Finally, we would consider the scenario of a bunch of network functions from multiple clients to deploy on a cluster of machines, to assess whether it should use a layered approach, or use joint optimization.

5.3.2 VM-based or container-based environments

In parallel, we will follow a similar approach for VM-based or container-based environments. Here we will first consider the deployment of a service chain on


a single machine, where all the VMs/containers are connected through a software switch (DPDK-enabled version of Open vSwitch). The challenge here is resource partitioning between the various components of the chain. Then we will move to larger, more complex scenarios where a number of network functions are distributed over a cluster of nodes, each capable of running a number of VMs.

6 Conclusion & Future work

This deliverable described progress that has been made on the function allocation optimisation task since the last I5.1 deliverable. Click-based environments have been further improved, with support for more heterogeneity and flow-processing, improving performance in both cases. The network performance framework (NPF) allows for easy deployment and analysis of network experiments, and has already been used to a great extent to characterize the behaviour of various RFB implementations in Click-based environments. Finally, the utility functions framework offers a flexible way to evaluate network application performance, based on many different source of information and various distinct KPIs, in such a way that it can either be used directly, when possible (*i.e.* simply by ranking different deployment alternatives, based on their utility), or as a reward to train a machine-learning based system.

In the future, we will focus both on the implementation and evaluation of our machine-learning based approach to solve the function (or resource) allocation problem, both in Click-based and VM-based (or container-based) environments. We will also further investigate and validate the utility function framework. Finally, we will work in close collaboration with task 5.3 for the definition of the APIs which will be used to query and/or deploy the optimised function allocation.

7 References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] H. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. Kaafar, and L. Mathy. SplitBox: Toward Efficient Private Network Function Virtualization. https://arxiv.org/abs/1605.03772. 2016.
[25] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2015.
[26] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In Eurocrypt, 2004.
[27] J. H. Cheon, K. Han, C. Lee, H. Ryu, and D. Stehle. Cryptanalysis of the Multilinear Map over the Integers. In Eurocrypt, 2015.
[28] J.-S. Coron, T. Lepoint, and M. Tibouchi. Practical Multilinear Maps over the Integers. In CRYPTO, 2013.
[29] N. A. Jagadeesan, R. Pal, K. Nadikuditi, Y. Huang, E. Shi, and M. Yu. A Secure Computation Framework for SDNs. In HotSDN '14, 2014.
[30] A. R. Khakpour and A. X. Liu. First Step Toward Cloud-Based Firewalling. In SRDS, 2012.
[31] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In NSDI, 2016.
[32] L. Melis, H. J. Asghar, E. De Cristofaro, and M. A. Kaafar. Private Processing of Outsourced Network Functions: Feasibility and Constructions. In ACM Workshop on SDN-NFV Security, 2016.
[33] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In SIGCOMM, 2015.
[34] J. Shi, Y. Zhang, and S. Zhong. Privacy-preserving Network Functionality Outsourcing. http://arxiv.org/abs/1502.00389, 2015.
[35] The Linux Foundation. Packetgen. http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen.
[36] A. Waterland. Stress. http://people.seas.harvard.edu/~apw/stress/.
[37] W. Norcott. IOzone. http://www.iozone.org/.
[38] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Forwarding path architecture for multicore software routers. In ACM PRESTO, 2010.
[39] 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.
[40] 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.
[41] 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
[42] John Wilkes. Utility Functions, prices and negotiations. 2008.
[43] 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.
[44] 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.
[45] 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.
[46] 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.
[47] 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.

[48] 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.
[49] 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.
[50] 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.
[51] 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.
[52] Cisco. Snort: Network Intrusion Detection & Prevention System. http://www.snort.org/.
[53] HAProxy: The reliable, high performance TCP/HTTP load balancer. http://www.haproxy.org/.
[54] NGINX Inc. NGINX: High performance load balancer, web server & reverse proxy. https://www.nginx.com/.
[55] 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.
[56] 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.
[57] 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.
[58] NPF: Network Performance Framework. https://github.com/tbarbette/npf.
[59] Superfluidity. I5.1: First report on function allocation algorithms implementation and evaluation. 2016.
[60] Superfluidity. D3.1: Final system architecture, programming interfaces and security framework specification. 2016.