Future Service Oriented Networks
www.fusion-project.eu

Deliverable D3.3

Final node design, algorithms and protocols for service-oriented network management and simulation results

Public report[1], Version 2.0, June 2016

Authors

UCL	David Griffin, Miguel Rio, Khoa Phan, Elisa Maini
ALUB	Frederik Vandeputte, Luc Vermoesen
ORANGE	Dariusz Bursztynowski
SPINOR	Folker Schamel, Mahy Aly
IMINDS	Piet Smet, Pieter Simoens

Reviewers

Abstract In this final deliverable of WP3, we present a final update on the specification, design and implementation of service orchestration and management in FUSION. In this deliverable, we first motivate the need for a FUSION orchestration layer, and present an overview of the final key FUSION orchestration concepts, high-level orchestration architecture and novel contributions. Then we present various placement algorithms for handling hierarchical orchestration, composite service graphs and on-demand service deployment. Next, we discuss our work on making a heterogeneous cloud platform a reality, presenting a lightweight composite service model, describe our work on application and system profiling and monitoring, as well as a possible design. Finally, we also present the final design and implementation of a prototype FUSION orchestration layer and each of the key components. The FUSION orchestration protocol specifications are grouped in a separate document I2.1: final specification of FUSION interfaces.

Keywords FUSION, service management, hierarchical orchestration, design, interfaces, algorithms, heterogeneous execution environments, light-weight virtualization

[1] The public version of this report has some sections suppressed as they are currently under review for inclusion in scientific publications.

Project funded by the European Union under the
Information and Communication Technologies FP7 Cooperation Programme
Grant Agreement number 318205

Revision history

Date	Editor	Status	Version	Changes
09/09/2015	F. Vandeputte	Initial Version	0.1	Initial ToC
20/01/2016	F. Vandeputte	Initial Version	0.1.1	Revised ToC
30/03/2016	F. Vandeputte	Draft	0.2	Initial Draft
13/04/2016	F. Vandeputte	Draft	0.3	Updated Draft
29/04/2016	F. Vandeputte	Stable Version	0.4	Stable Draft
04/05/2016	F. Vandeputte	Stable Version	1.0	Stable Version
20/07/2016	F. Vandeputte	Final Version	2.0	Final Version

GLOSSARY OF ACRONYMS

Acronym	Definition
2D	2-dimensional
API	Application Programming Interface
AUFS	Advanced Unification File System
AVX	Advanced Vector Extensions
BGRA	Blue-Green-Red-Alpha channels
BW	Bandwidth
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
COTS	Common Of The Shelf
CPE	Customer Premises Equipment
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
D	Domain
DC	Data Centre
DCA	Data Centre Adaptor
DCTCP	Data Centre TCP
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
E2E	End-to-End
EC2	Amazon Elastic Compute Cloud
EPG	Electronic Programming Guide
ETCD	A highly available distributed key value store for shared configuration
EZ	Execution Zone
FPS	Frames Per Second
FUSION	Future Service Oriented Networks
GA	Genetic Algorithm
GCE	Google Cloud Engine
GHz	GigaHertz
GPU	Graphical Processing Unit
HTTP	Hypertext Transfer Protocol
HW	Hardware
HWM	High Water Mark
I/O	Input/Output
ID	Identifier
ILP	Integer Linear Programming

IP	Internet Protocol
ISC	Inter-Service Communication
IT	Information Technology
IVSHMEM	Inter-VM Shared Memory
JBOF	Just a Bunch Of Flash
JVM	Java Virtual Machine
KSM	Kernel Samepage Merging
KVM	Kernel-based Virtual Machine
MB	MegaByte
MTBF	Mean Time Between Failure
MTU	Maximum Transmission Unit
NIC	Network Interface Controller/Card
NP-hard	Non-Deterministic Polynomial-Time hard
NUMA	Non-Uniform Memory Architecture
NVM	Non-Volatile Memory
OCP	Open Compute Project
OS	Operating System
OVS	Open vSwitch
PCA	Principal Component Analysis
PCI(e)	Peripheral Component Interconnect (Express)
POSIX	Portable Operating-System Interface
PPC	Performance-Predictability Curve
QoE	Quality of Experience
QoS	Quality of Service
QPI	Quick-Path Interconnect
RAM	Random-Access Memory
RD	Resolution Domain
REST	Representational State Transfer
RT	Real time
RTT	RoundTrip Time
SHM	Shared Memory
SLA	Service Level Agreement
Speedus EP	Speedus Extreme Performance
SR-IOV	Single-Root IO Virtualization
SSD	Solid State Disk
SSE	Streaming SIMD Extensions
STB	Settopbox
SW	Software

TB	TerraByte
TCO	Total Cost of Ownership
TCP	Transmission Control Protocol
VM	Virtual Machine
VR	Virtual Reality
YUV	Particular Video Color Space
ZMQ	Zero Message Queue

EXECUTIVE SUMMARY

This document is a public deliverable of the “Future Service-Oriented Networks” (FUSION) FP7 project. It focuses on the algorithms, protocols and functionality of the service management and orchestration layer in FUSION, including the design and implementation of all the FUSION components implementing this service layer.

The key challenges for this work package are twofold. A first challenge is how to build a composable and scalable service layer for efficiently managing large-scale personalized services across distributed heterogeneous execution nodes. A second challenge is developing efficient and scalable service placement and scaling algorithms for optimally deciding on when and where to deploy particular service components, based on service demand, service requirements and platform capabilities.

The service management layer is conceived as a set of orchestration domains. In each orchestration domain, there is a logically centralized domain orchestrator where service providers can register and manage their services, and where services are globally managed within a domain. This includes the high-level placement and scaling of the services in the various execution zones that are managed by the domain. Each execution zone is managed by a FUSION Zone Manager, which handles all local deployment, monitoring, load balancing and scaling management. This decoupling ensures a scalable service layer in which all local management details are handled locally and all global decisions are made at the domain level.

Within an execution zone, we also distinguish between the higher-level FUSION management and orchestration and the lower-level deployment and management of service instances on either physical or virtual resources. In FUSION, we envision that FUSION Zone Managers could be deployed on either bare physical environments or on cloud infrastructures, which may be managed by a third party. The former type allows for a much higher and finer degree of control of the underlying resources but requires direct access and low-level management of FUSION. The latter type allows FUSION to delegate most of the lower-level deployment details to the underlying cloud platform, but obviously allows for less fine-grained control. To be able to abstract this varying degree of control and low-level management, we designed a Data Centre Adaptor layer, to clearly distinguish the higher-level FUSION zone management functions from the lower-level deployment functions.

This deliverable is the third public Deliverable for Work Package 3 and extends, refines and finalizes the various concepts, design and high-level inter-layer protocol interfaces that were already described in Deliverables D3.2. This deliverable contains four main parts. In the first part, we restate the relevance and need for a distributed heterogeneous cloud platform such as FUSION for managing demanding interactive (near) real time cloud services, and provide an overview of all key FUSION orchestration concepts and novel contributions since D3.2.

The second part of this deliverable then focuses on placement algorithms for hierarchical high-level orchestration as well as composite services, and discusses an algorithm for smartly provisioning container image layers across all execution nodes to allow for fast on-demand service deployment, which is especially important in case of long-tail services in a very distributed cloud environment where the location of the service with respect to the client is crucial for proper QoE.

The third part of this deliverable then focuses on the heterogeneity of the cloud infrastructure, both within a data centre (or execution zone), as well as across data centres. We motivate a lightweight composite application service model and present a number of profiling and monitoring mechanisms for more optimally deploying these sensitive applications on the various heterogeneous runtime environments. We refine two key FUSION orchestration concepts, namely session slots and evaluator services, and analyze their overhead and benefits for FUSION. We present a visualization mechanism for representing the trade-off between performance and predictability and study various inter-service communication mechanisms for efficiently implementing this lightweight composite service model. All

this is summarized in a description of a heterogeneous cloud platform that automates the entire deployment and configuration optimization process.

Finally, we present and discuss the final high-level design of the FUSION orchestration plane, zooming in on the various orchestration layers, including their high-level designs as well as the status of their prototype implementation.

TABLE OF CONTENTS

1. INTRODUCTION: FUSION ORCHESTRATION AND MANAGEMENT

In this introductory chapter, we start by motivating the need for a distributed and heterogeneous FUSION orchestration and management plane, followed by an overview of all key FUSION orchestration concepts that have been introduced in previous FUSION deliverables as well as additional concepts. Next, we discuss the final high-level FUSION orchestration architecture, and end this chapter with a summary of the key new contributions made in this Deliverable. Each of these contributions will be elaborated in detail in the subsequent chapters.

1.1 Motivation

In FUSION, we envision new classes of applications that could benefit from running in the cloud. Examples include real-time media applications such as virtual reality, which requires very high bandwidth and ultra-high resolution, rendered at very high frame rates [Oh16], and ultra-low-latency [We16]. Other real-time media applications include multi-player gaming, EPGs, etc. But also IoT applications, producing plenty of raw data to be processed an analyzed, can benefit from the cloud.

Due to their inherent nature of being (1) very demanding, (2) high bandwidth and/or (3) ultra-low latency, these applications cannot cost-effectively be running from a general-purpose central cloud. Instead, they require a more distributed and heterogeneous cloud environment to meet their requirements and to be cost-effective. The need for distribution is driven mainly by the bandwidth [VER14] and low-latency requirements [We16], whereas the heterogeneity is mainly driven by the specific and demanding computational requirements of these new classes of applications (see also Section 3.1).

The distributed and heterogeneous nature of such cloud environment puts however special requirements onto the placement and scaling of these services, which need to be taken into account for deciding both *when* and *where* to (pre)deploy these services or service components across such cloud environment.

With respect to the *when*, we distinguish between two main classes of services. The first class of services is the set of popular services for which it makes sense to already pre-deploy a number of instances across a distributed cloud environment such as FUSION, based on expected/predicted demand patterns in particular regions. The second class is the class of services for which it is nearly impossible to determine/predict a good location on where to pre-deploy an instance in advance. This includes both long-tail services, but also services that need to be extremely close to end user, and as such they exhibit the characteristics of a long-tail service in the local area (though they are not necessarily a global long-tail service). As such, to be able to handle both classes of services, we have studied placement and deployment scenarios and algorithms for both cases, namely the *pre-deployed* and *on-demand* deployment scenarios.

With respect to the *where*, we have studied on how to most effectively take into account service requirements, networking characteristics as well as execution node capabilities for optimizing service deployment across different execution zones (i.e., global placement) as well as within execution zones (i.e., local placement). Two key concepts here are the utility function and evaluator services.

In a global distributed FUSION cloud environment, another key aspect is scalability of the orchestration layer. As such, in this Deliverable, we also studied the impact of a hierarchical orchestration domain for efficiently managing services across this multitude of smaller and/or larger scale execution zones and geographical domains.

Additionally, to be able to optimally leverage the fundamental capabilities provided by FUSION, we also advocate a composite (micro-)services model (see Section 3.2), where application services in fact are a graph consisting of multiple/many smaller service components with both “weaker” and “stronger” links in between them. Strongly-linked service components will typically be deployed within the same execution zone using specialized inter-service communication protocols to minimize communication overhead, but each running in their own optimized heterogeneous SW/HW environment. Indeed, in FUSION, we envision a more heterogeneous nature of SW/HW execution environments to more efficiently deploy these demanding services, see Section 3. Weakly-linked service components on the other hand need to be optimally deployed across different zones, taking into account all networking aspects. As such, we also studied composite service deployment algorithms in the context of FUSION.

A key principle w.r.t. composite service deployment is that we envision a dynamic and flexible deployment and interconnectivity of instances of these services. Rather than fixing the service graph at deployment time, we mainly ensure that feasible service graphs are available but leave the final decision up to the FUSION resolution plane, allowing more flexibility and reuse. This model was already discussed in detail in Deliverable D3.2.

To cost-efficiently manage these demanding services, we envision an automated heterogeneous cloud platform, consisting of a wide range of general-purpose and special-purpose hardware resources, combined with a set of OS configuration optimizations for efficiently managing these heterogeneous resources and application services, and on top a an automated cloud platform optimization layer that can automatically optimize how services are deployed on this heterogeneous cloud platform. For this, we envision an offline and online application and resource profiling and monitoring framework to be the back bone of this continuous self-optimizing cloud platform. As such, in this deliverable (as well as previous WP3 deliverables), we also studied various profiling and monitoring mechanism and investigated how they could contribute to this platform.

The high-level FUSION orchestration architecture itself is based on an overlay approach, dynamically leveraging the multitude of available cloud infrastructures from different providers that are already out there. This facilitates the deployment of a global distributed FUSION platform, rather than having to manage and deploy your own physical data centres across the globe.

In conclusion, in FUSION, we propose an end-to-end hierarchical distributed heterogeneous cloud orchestration platform that can be deployed on top of existing cloud infrastructures for enabling next-generation highly personalized and often resource-demanding cloud services. This deliverable presents our contributions for achieving this vision. To this end, we first start by restating the key FUSION orchestration concepts and enablers that have been introduced as part of WP3.

1.2 Key FUSION Concepts

In this section, we summarize the key FUSION orchestration concepts, terminology and enablers that were created and/or used as part of WP3. More details on each concept of term can be found in previous deliverables as well as this deliverable.

1.2.1 Architectural Concepts

These will be elaborated in further detail in Section 1.3 and especially in Section 4.

- **Domain (orchestrator)**

  Top-level (logically) centralized orchestration service that manages all registered FUSION application services across a set of distributed execution zones.

- **Resolution domain (orchestrator)**

  Intermediate level orchestration service that manages a subset of registered FUSION services across a subset of execution zones.

- **Zone (manager)**

Local management and orchestration service that is responsible for the high-level lifecycle management of FUSION services in one (or more) local data centres.

- **Data centre adaptor (DCA)**

  Translating and/or gluing layer in between a FUSION execution zone and the actual physical data centre. In FUSION, we envision that any portion of any cloud or data centre could be made FUSION-enabled by deploying a FUSION zone manager on top of that data centre. The DCA layer is an adaptor layer to manage the translation between high-level FUSION APIs and map them onto the actual DC APIs.

- **Application service**

  A service that registered in a FUSION domain by an application service provider. Once the service is registered, the FUSION domain will automatically manage its lifecycle across its distributed cloud environment, based on user demand, resource availability and preferences/policies set by various entities (e.g., service provider, domain orchestrator, etc.).

- **Application service provider**

  Internal or external entity that registers application services in a FUSION domain.

- **Client**

  Piece of software (e.g. operated by end users or other services), that connects to an application service deployed in FUSION.

1.2.2 Functional Concepts

- **Evaluator service**

  To allow service providers to have control over where application service instances are being deployed, the concept of an evaluator service was introduced in FUSION. This is a service specific probe that is deployed prior to the application itself and that can assess how suitable that particular runtime environment is for running some service. This is especially important in a dynamic heterogeneous cloud environment such as FUSION.

- **Session slots**

  Another key FUSION concept is that of a session slot. A session slot is a lightweight abstract application metric for measuring the resource utilization and availability for a particular service instance. Specifically, it is a number representing the number of parallel sessions that can (still) be supported with sufficient QoS (i.e., deadline misses, latency, jitter, etc.) for a particular instance. For example, if an instance is reporting 4 available session slots, this means that it has still sufficient resources for handling 4 clients in parallel for the duration of the session.

- **Service aliasing (or multi-configuration service instances)**

  In Deliverable D3.2, we introduced the concept of service aliasing, or multi-configuration service instances. Basically, it allows for particular instances to be announcing (different amounts of) session slots for different service names. This allows to reuse the available resources allocated to particular service instances for multiple types of service instances, resulting in a higher reuse and lower fragmentation. A typical example is a gaming service instance that can host both premium service sessions as well as basic service sessions, though perhaps with a different amount of available slots, rather than having a separate set of instances for handling the premium and basic gaming sessions. Using different service names on the other hand allows to set different placement, selection and scaling policies for different services. For example, the utility function and resolution requirements may be much more stringent than for a basic gaming service, though they reuse the same physical instances.

1.2.3 Key enabling technologies & methodologies

- **Lightweight containers**

  Lightweight application isolation and virtualization mechanism provided by the host OS. In this model, each application is packaged and deployed separately on top of a single host OS, who will isolate them from each other via the kernel. This is in contrast to standard virtualization techniques, where entire machines, containing also the application(s) and OS are packaged and deployed as a virtual machine on top of a hypervisor, whose responsibility is to schedule these virtual machines on a single physical machine and manage its resources.

- **Microservices**

  Application model where an application is decomposed into a graph of loosely coupled independent but interacting services, each having a single specific function and API. This is in contrast to a single fast service.

- **Microservers**

  Hardware design where a physical server is decomposed into a set of smaller and often dynamically composable service components, connected to each other via a specific interconnect. Microservers typically focus on energy-efficiency and a higher utilization of resources. This hardware design is in contrast to the existing standard fat cloud nodes, containing a fixed mixture of fat CPUs, massive amounts of RAM memory, storage and networking.

- **Heterogeneous cloud**

  Rather than having a standard cloud with a small number of very similar cloud nodes that are configured in the same way, in FUSION, we envision a truly heterogeneous cloud (and platform), where a wide and dynamic range of different resources and accelerators are available and optimized via a dynamically reconfigurable software platform that is responsible for optimally deploying demanding applications on top of the heterogeneous hardware.

- **Hierarchical and distributed orchestration**

  For scalability as well as efficiency, we opted for a hierarchical and distributed orchestration layer, where there are multiple layers of orchestration services on top of each other as well as next to each other, each performing only a subset of the total orchestration function, but coordinated via a specific hierarchical structure.

1.3 Final High-Level FUSION Orchestration Architecture

In Figure 1, the final high-level FUSION orchestration architecture is depicted. A key new element compared to the previous high-level orchestration architecture, is the addition of one or more intermediate ‘resolution’ orchestration domain layers to be able to better cope with scalability and local optimizations based on local-only data. See Section 2.1 for more details.

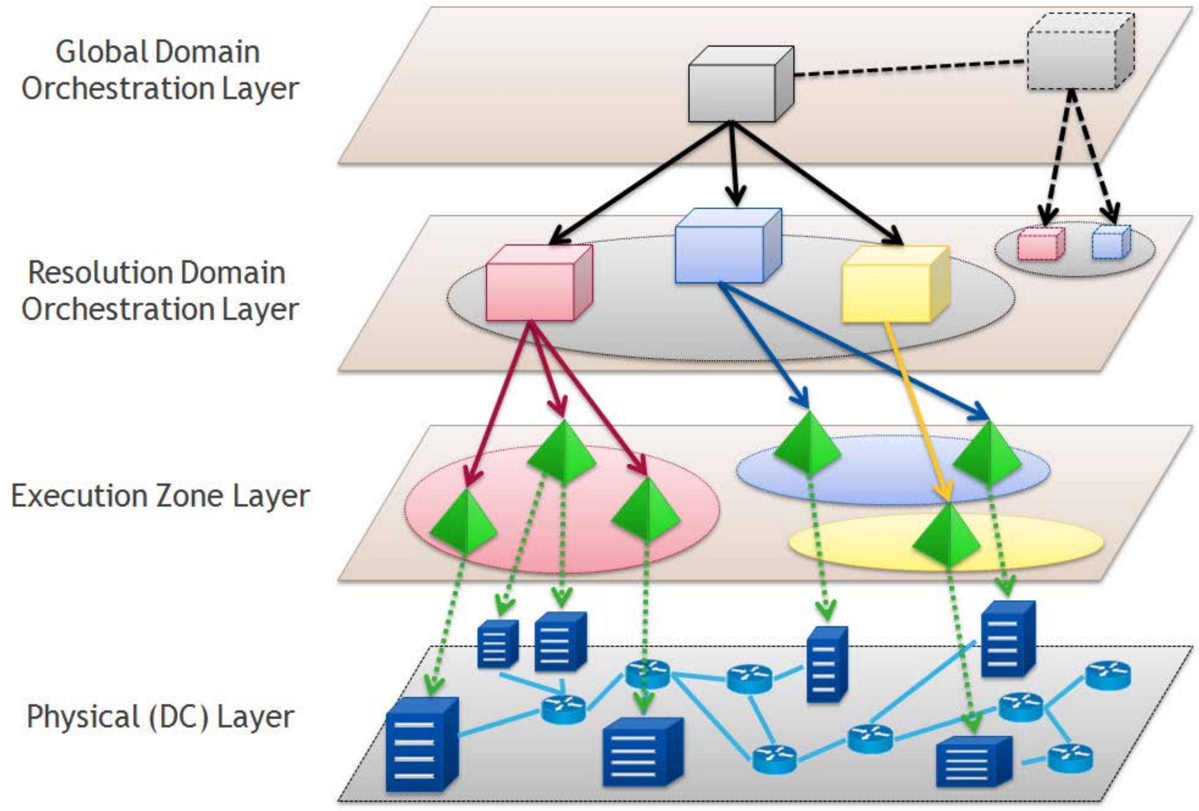

Global Domain Orchestration Layer
Resolution Domain Orchestration Layer
Execution Zone Layer
Physical (DC) Layer

Figure 1 – Final high-level FUSION orchestration architecture.

In general, the high-level FUSION orchestration architecture consists of 3 layers on top of the physical layer. The physical layer is the layer where the physical data centres and networks are. These physical data centres may or may not have their own data centre management layer; in FUSION, we envision an overlay approach, allowing to deploy FUSION orchestration on top of any existing data centre. As will be discussed in Section 4, we envision an additional (but hidden) data centre adaptor layer to efficiently cope with interoperability.

FUSION services are being deployed and managed on these data centres via execution zones. In this layer, (parts of) one or more data centres are being managed by execution zones for deploying FUSION services. These execution zones however typically are only responsible for local orchestration of FUSION services on top of one or more local data centres or nodes. They do not have a global view or management responsibility. This global placement and service management is handled by the domain orchestration layers.

In the original architecture, we envisioned only a single domain orchestration layer. However, for scalability and efficiency, we introduced the possibility of one or more intermediate subdomain orchestration layers. In the figure, we showed a single subdomain layer, which we called the resolution domain orchestration layer, as these subdomains coincide in this example with the resolution planes. At the top level, there is then a logically single global domain orchestrator (per FUSION domain provider) that has the global overview. Service providers register their services typically at this global orchestrator, who may then pass on the management to one or more intermediate domain orchestrators, though the global orchestrator may keep the global overview. There may be multiple independent global FUSION domain providers, each managing their own subdomains, execution zones, etc.

1.4 Key New Contributions

In addition to the WP3 contributions that were already presented in the previous Deliverables D3.1 and D3.2, this Deliverable makes the following new contributions:

- We expand the FUSION orchestration architecture to also incorporate hierarchical orchestration to better cope with the scalability of placing, deploying and managing services across an entire globe of execution zones.
- We present a novel hierarchical placement algorithm to efficiently handle with a hierarchical orchestrator, balancing between efficiency and optimality.
- We propose and evaluate various placement algorithms for various types of composite service graphs.
- We describe fast on-demand service provisioning algorithms to be able to provision and deploy services within a short amount of time, irrespective of their location or popularity, and with a constrained amount of storage (i.e., cost).
- We discuss the benefits of a lightweight composite application service model in a heterogeneous cloud platform
- We present a number of application and system profiling and monitoring techniques for characterizing and evaluating applications running in particular runtime environments.
- We present a visualization technique for representing the trade-off between performance and predictability of a particular application and/or system.
- We expand the FUSION session slots concept and analyze its overhead and benefits.
- We expand the FUSION evaluator services concept and also analyze its overhead and benefits.
- We evaluate various inter-service communication mechanisms for efficiently exchanging large amounts of data in between these lightweight service components.
- We present the results of a study on extreme service node density.
- We present an updated version of a possible heterogeneous cloud platform design for efficiently and flexibly managing the heterogeneity of services and underlying software and hardware infrastructures.
- We present a final version of a FUSION orchestration architecture and present for each orchestration layer their final high-level design as well as prototype implementation status.
- We updated the protocol specifications for all components, and implemented them into the prototypes.

Each of these contributions will be presented, motivated and evaluated in the following chapters.

2. DOMAIN ORCHESTRATION AND PLACEMENT ALGORITHMS

This section covers key design decisions and corresponding algorithms related to hierarchical orchestration, placement algorithms for composite services as well as on-demand service provisioning.

2.1 Hierarchical Orchestration

2.1.1 Hierarchical service placement

In our studies on service placement so far we have considered a centralised model where the service orchestrator has a detailed view of all EZs and all forecasted user demands for a particular service and it optimises the placement of service instances in the EZs to maximise total utility within cost constraints set by the service provider.

It may be impractical, for scalability reasons, for a globally centralised placement algorithm to maintain detailed knowledge of all users and all EZs and so here we investigate a hierarchical solution where the overall orchestration domain is split into geographical sub-domains. In this model the high-level orchestrator has limited visibility of EZs and user demands within a sub-domain – it sees only the aggregate of user demands and the aggregate of EZ capacities within a particular sub-domain. The high-level orchestrator places service instances at the coarse granularity of sub-domain only and subsequently each sub-domain orchestrator undertakes a further placement algorithm with the scope of that sub-domain only to determine in which specific EZs what quantity of service instances should be placed to supply the required number of session slots to meet the specific detailed demand pattern of user requests within that sub-domain.

There are many ways of sub-dividing an overall orchestration domain into sub-domains. One option is to map sub-domains onto the same geographical area covered by resolution domains: the entity responsible for resolving user requests to EZs with available session slots. Equating sub-domains for orchestration and service placement purposes with resolution domains isn't essential as other coarser or finer grained sub-domains could be considered. However, in the rest of this section we assume that the lower-level orchestration domains have been mapped onto resolution domains and the term *resolution domain* is used to mean the lower-level sub-domain for orchestration and service placement. Note that is also possible to consider multiple hierarchical levels of service orchestration and placement; however, we only model two levels in the analysis described in this section.

In summary, as each lower-level domain is treated as a black box with respect to the high-level orchestrator the overall service placement problem can be divided into smaller units – one at the high level working at coarse granularity and several (one per sub-domain) operating at a lower level with more detailed information but with limited geographical scope. In this way the optimisation algorithms can be executed with reduced quantities of information, increasing scalability. In this section we investigate how the performance of hierarchical service placement compares to the centralised model.

2.1.2 Problem description and methodology

This section presents the modelling approach to analyse and evaluate the performance of our framework. In particular, we split the whole world into eleven geographical regions, based on the continents, but with larger continents split into two or three regions to make the population of each region roughly the same. The resulting regions are: Western Europe (EUW), Eastern Europe (EUE), Central Asia (ASC), Southern Asia (ASS), Pacific Asia (ASP), Africa (AFR), Northern North America (NAN), Southern North America (NAS), Eastern South America (SAE), Western South America (SAW) and Oceania (OCE). We further sub-divide each region into N low-level orchestration sub-domains, which are termed resolution domains in the remainder of this section, but it should be born in mind that although the low-level orchestration sub-domains have been mapped to resolution domains here this isn't the only granularity of orchestration sub-domain that can be considered. For our simulation models we select the N most populated cities in each region which become the *centroids of the* *resolution domains in that region*. Users and EZs are mapped to their geographically closest centroid, and are said to *belong* to that resolution domain.

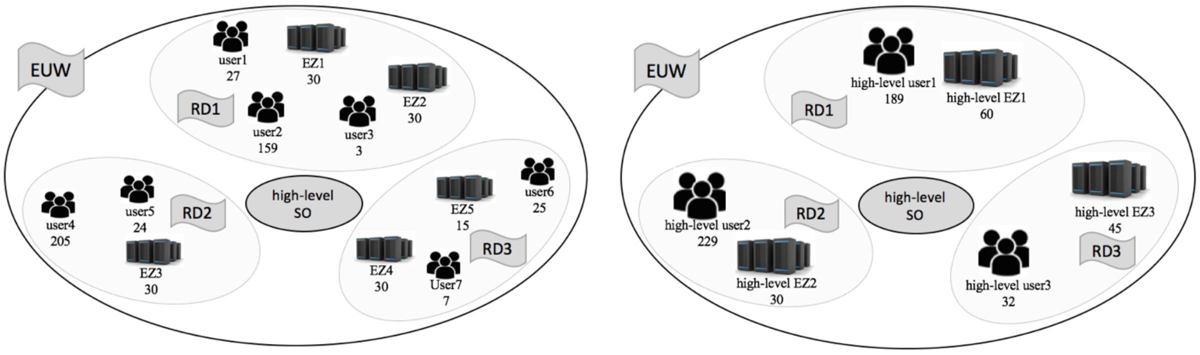


Figure 2 – EUW region split in three RDs (RD1, RD2, and RD3)

Each resolution domain (low-level orchestration sub-domain) consists of a number of EZs and users at specific locations. But, from the perspective of the high-level orchestrator the capacities of the EZs are aggregated into a single logical EZ (termed *high-level EZ*) located at the centroid of the resolution domain and the individual users are modelled as a single group of users (termed *high-level user*), also located at the centroid. As an example, shows the Western Europe (EUW) region split in three resolution domains (RD1, RD2, and RD3). On the right-hand part of the figure the individual EZs and users are grouped into their high-level counterparts, which are located at the resolution domain's centroid.

The high-level orchestration algorithm only sees the set of high-level EZs and high-level users, located at the centroids of the lower level sub-domains. It runs the centralised optimization (described in D3.2 previously) which is solved to find high-level EZs should deploy what quantity of service instances to meet the predicted demand of the high-level users. shows an example of the placement decision made by the high-level orchestrator. In each resolution domain some service demand is allocated to the local high-level EZ (*local-to-local requests*), i.e. the high-level orchestrator has matched local demand to local EZs, some are allocated to a remote EZ (*local-to-remote requests*) or come from a remote user (*remote-to-local requests*). When there are not enough resources in terms of EZ capacity to match user requests to EZs within the maximum utility for the service or if the cost of the solution would exceed the cost constraints, user requests are *blocked.*

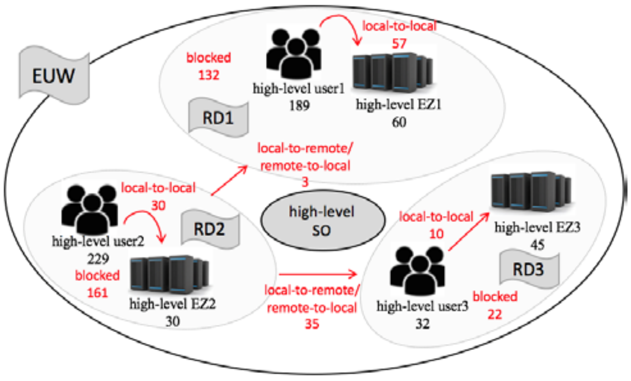


Figure 3 – Solution of the centralized optimization algorithm.

Using the hierarchical approach, the optimisation problem has been split into a number of sub-problems, each of which can be undertaken with a smaller quantity of information, reducing the time to find a feasible solution. Since the high-level orchestrator does not know detailed information for each sub-domain in terms of the individual EZs and individual users belonging to each domain it is unable to determine the precise quantity of session slots to be allocated to EZs and which of these will be allocated to which individual users. Hence the first task of the low-level optimisation algorithms is to map the output of the high-level placement into a more detailed input for the low-level placement optimisation function. A post-processing phase is locally used in each low-level orchestrator to attribute the total local-to-local, local-to-remote and blocked demand to individual low-level users within that sub-domain and how the total remote-to-local requests are allocated to individual low-level EZs in that sub-domain.

In particular, local-to-local requests and blocked requests are simply allocated proportionally to the initial quantity of individual user demand. For instance, consider RD2 where the total blocked requests are 161; after allocating these proportionally 144 requests from user4 are blocked and 17 from user5. Similarly, the total local-to-remote requests are 38 (the sum of those to RD1 and RD3), these are split between the low-level users such that 34 are from user4 and 4 from user5. After having attributed the blocked requests and the local-to-remote requests to each low-level user, the remaining service requests are are 34 for user4 and 4 for user5, which are the total local-to-local requests for RD2. Local-to-remote request for the RD2 are remote-to-local requests for other resolution domains; in particular, 3 remote-to-local requests for RD1 and 35 for RD3. Such requests must be allocated to the EZs belonging to RD1 and RD3, respectively. We choose to allocate proportionally, for instance for RD3 we consider a remote user with 35 requests from RD2 and for RD1 a remote user with 3 requests from RD2. Local-to-local service requests are then allocated using the placement optimization algorithm running in each low-level sub-domain ().

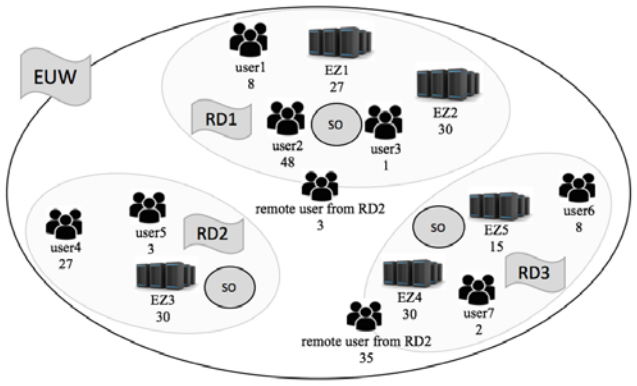


Figure 4 – Service requests and available session slots after the post-processing phase.

Now that the aggregate local-to-remote and remote-to-local demands which had previously been determined by the high- level placement algorithm have been allocated to individual EZs and users by each low-level optimization algorithm, this is used as input to the full placement optimization algorithm running in each sub-domain.

2.1.3 Experimental setup

Simulations use real data to infer the geographic location of users and data centres. In order to create the simulations, the model includes: (i) a distribution of users across the surface of the earth; (ii) a distribution of data centres around the globe; (iii) an approximation of service deployment cost which is based on regional Amazon EC2 costs. The dataset includes 2048 data centres distributed in 525 cities, we group the data-centres in the same city and define this as the execution zone in that city. We define the capacity of each execution zone considering a model described in the next section. The latency between users and execution zones are calculated based on the Haversine distance, the shortest distance between two points around the planet's surface. We map the distance to latency figures according to the formula defined in [LANDA13] which modelled distance to latency based on real data measurements. We define three latency thresholds R_{min} = 20 ms, R_{med} = 100 ms, and R_{max} = 150 ms (see

D3.2 for a description of the utility function and its latency thresholds). The cost of the service deployment at each execution zone is set proportionally to the cost of VMs in Amazon EC2.

Mismatch between supply and demand

For the simulations, we first set the maximum capacity of each EZs so that it is sufficient to serve all demands according to users selecting the closest local EZ. In particular, we use supply/demand = 130% and we call that configuration *perfect allocation*. Next we create different level of mismatch between supply and demand varying a parameter "*N% rand*". That means that from each EZ we move N% of capacity (modelled as session slots) to other randomly selected EZs. In particular, we show evaluation results for "0% rand." (which is equivalent to the perfect allocation), "40% rand.", and "100% rand.".

2.1.4 Simulation results

We show in Figure 5 a comparison between the CDF latency of the hierarchical and the centralised algorithms with different mismatch levels between supply and demand ("X% rand."). In particular, we first run the centralised model but without the capacity to find a *perfect placement* solution assume that we do not have any resources' constraints. Next, we create different levels of mismatch between supply and demand by varying a parameter "X% rand.". This is for each EZ we remove the X% of its session slots from the *perfect placement* configuration. Then, we mix the removed session slots of all EZs and scatter them uniformly in all EZs. This guarantee that the total session slots of EZs in all cases (*perfect placement* and "X% rand.") are the same. "0% rand." is equivalent to the *perfect placement* while in "100% rand." there is a uniform distribution of session slots between all EZs. The reason we vary "X% rand." is to test how the algorithms adapt with different configurations. X = 0% (Figure 5 (a)) corresponds to the "easy" case for the placement algorithms as it aligns with the *perfect placement* solution. On the other hand, X = 100% (Figure 5 (c)) is the "hard" configuration for finding a good placement solution. As shown in , we see that the gap between the hierarchical and the centralized algorithms increases when we increase X%. However, in general, the hierarchical algorithm performs well, close to the centralized one.

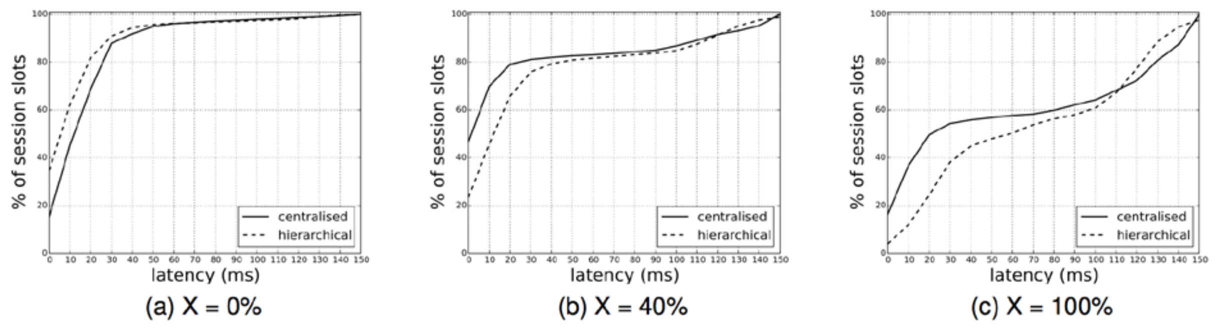


Figure 5: Comparison of centralized and hierarchical placement under different mismatch levels

In Figure 6 we show evaluating results for the hierarchical algorithm in terms of latency and utility with different value of "X% rand.". As mentioned before, increasing "X% rand." would result in worse QoS in term of latency and utility which can be seen clear in Figure 6. However, as our model try to maximize the total utility, the gap between the two cases X = 0% and X = 40% is small meaning that we still can find good solution even with 40% of mismatch configuration. In case, X = 100%, we do not have enough resource at EZs but the algorithm is successful to minimize the number of blocked user requests.

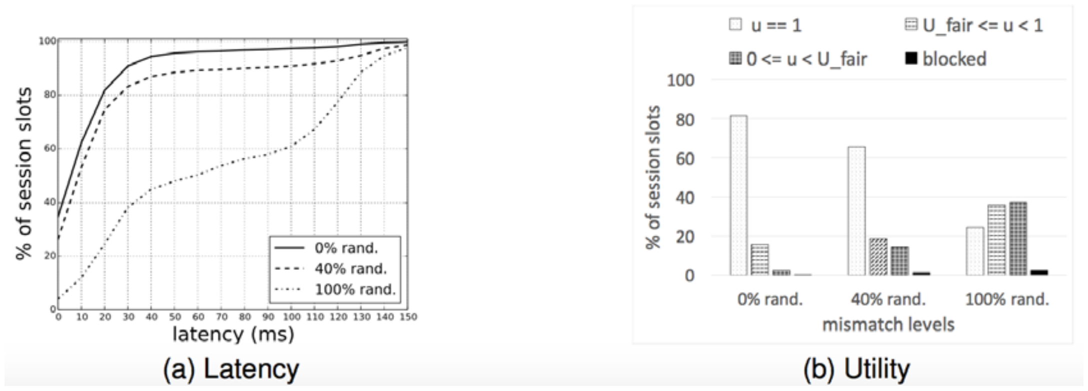


Figure 6: Latency and utility results for hierarchical placement under different mismatch levels

We show in Figure 7 the benefit of max-min fairness policy over all users. We test with different cost budget in the model. We first consider a minimum cost value that can find a feasible placement solution, then increase this budget by a factor shown in the x-axis of Figure 7. For each test, we check the latency of the worst user - the one that has the maximum latency over all users. As shown in Figure 7, with the max-min fairness constraint, the worst user still can get a good QoS (latency is less than 25 ms) while without max-min fairness, some users suffer from high latency (but it is always less than T_{max}). This confirms that with the max-min fairness, all users will have a better fair share of the resources at EZs.

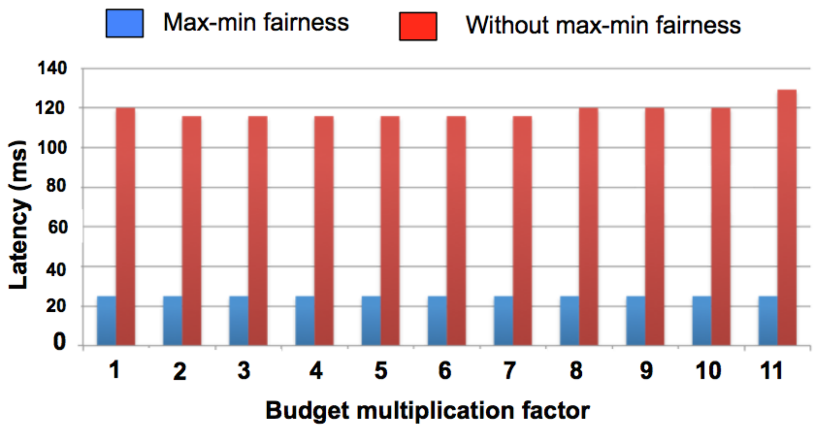


Figure 7: Latency of the worst user with and without max-min fairness

Considerations

The hierarchical approach tries to combine the advantage of both centralized and decentralized approaches. A single centralized orchestration has the advantage of ultimate level of visibility and control, as the placement algorithm has full visibility of individual users and execution zones. However, this solution is prone to scalability issues with millions of users using services across hundreds or thousands of execution zones. A centralised orchestration is impractical in real deployments, as a single global orchestrator would be required to collect information from all EZs and networks and also to model predicted demand from all users. On the other hand, with a completely decentralised approach, scalability can be handled more easily. However, this model implies that service providers need to register, deploy and manage their services with multiple providers in multiple locations, making the configuration and management of widely deployed services more complex. With the hierarchical model, the advantages of the decentralised model can be combined with global service orchestration.

2.2 Composite Service Placement

Although cloud paradigm provides several advantages in elasticity and cost efficiency, today's cloud infrastructures still do not match well with many innovative services' requirements. Therefore service placement policies will play a key role in delivering good quality of service (QoS) for users. There are many drivers for the service placement, including server resilience, network diversity, and proximity of servers to users. In this paper, we focus on deploying services closer to users to improve on QoS metrics like latency and/or throughput for all the users. We thus arrive in a situation where replicas of the same service are deployed in many data-centres - spread over the Internet. Service quality has two major sets of component metrics, relating to processing at servers and networking latency. The server placement system will have to take into account both computation and networking factors to optimize its performance.

Choosing a concrete service placement solution that is optimal with regard to QoS and cost constraints is an NP-hard problem. In this work, we focus on the composite service deployment problem which is more complex than the atomic service. A composite service includes several components located in different data-centres. When in use, we must determine an efficient way to connect those components as a right work-flow (Figure 8 - Figure 10). QoS of a composition is the aggregated QoS of the individual services according to the work-flow patterns. With the emergence of network function virtualisation (NFV) and distributed cloud paradigm, composite service paradigm becomes a wise choice which is flexible and reduces cost over the traditional atomic service model. For instance, an uncommon (rarely used) component can be deployed in one data-centre and will be shared by several users when needed to reduce the overall deployment cost. We will describe in detail the structures of composite services in Figure 8 - Figure 10.

2.2.1 Composite service structures

2.2.1.1 Snake or chain structure

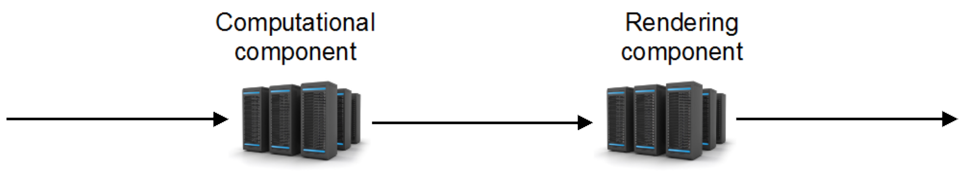


Figure 8 – Service chaining in online game

One of the most common structure of NFV composite services is the snake or the chain structure. As an example in Figure 8, two components in a online game can be deployed in different data-centres and are connected together as a chain when needed to provide necessary functionalities for the network. For this structure, the end-to-end latency will be accumulated from each hop in the chain (including network latency and processing delay at data-centres).

2.2.1.2 *Tree or parallel structure*

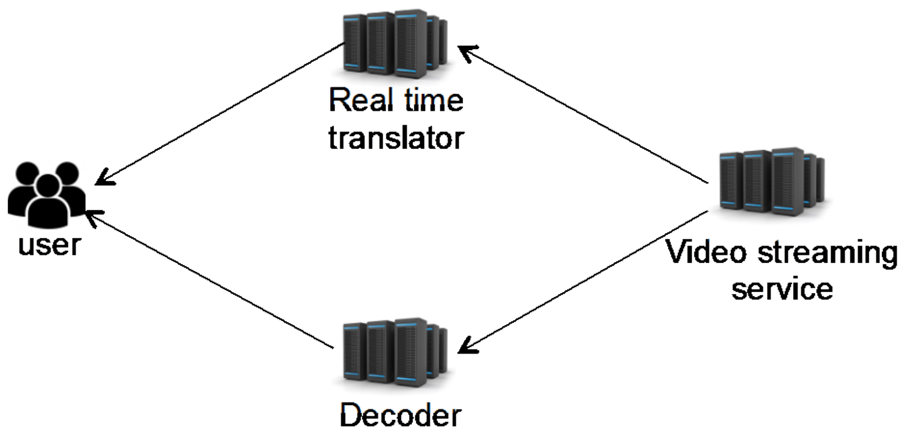


Figure 9 – Video streaming with translator and decoder.

We present in Figure 9 an example of the tree or parallel composite service structure. Assume that users would like to watch a video stream with a language audio and a specific codec which are not available. The streaming service therefore needs to send the audio to a real time translator component and a decoder which are deployed in different locations. The translated audio and decoded video are then forwarded to users as their requirements. The steps of translating and decoding are done in parallel and the end-to-end latency will be the longest one in the two branches.

2.2.1.3 *Circle structure*

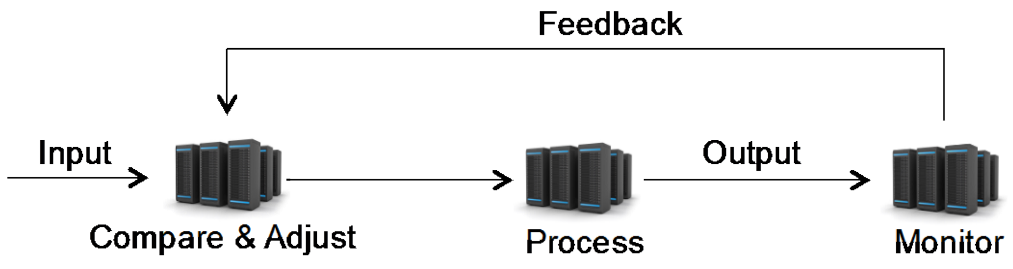


Figure 10 – Closed loop system.

The last structure we are presenting is the circle one with a loop. An example can be the closed loop system shown in Figure 10. It is similar to the chain structure except there is a loop to provide the feedback to make decision in next round. The end-to-end latency is accumulated for each hop.

Given the three basic structures, we can combine them to form any complex composite services. In this paper, we present both the exact and the heuristic algorithms for each of the basic composite structure. By maximizing the utility, we also consider several constraints for the composite service deployment problem:

- Fixed cost: the cost that we only pay once when the service is first deployed at a data-centre. This can be thought as the license cost for software installation or the cost to manage the service. Therefore, the fixed cost does not count on how many service instances we are using.
- Linear cost: this cost is proportional to how much resource the service needs. The more the service instances we deploy, the more resources are occupied, therefore we have to pay more. Latency: this includes both the network latency and processing time at data-centres. There is a trade-off between deploying services in a further geo-location but faster processing data-centre or choosing a closer but slower data-centre. Our algorithms will consider this trade-off in the optimization model. In addition, there are some services that require the connection

between users to the first hop component should not be too long. For instance, users should connect to a close rendering component in an online game service to achieve a smooth display processing. Therefore, along with the end-to-end latency, we also consider the first hop (or any hops) latency as a constraint when deploying a composite service.

The composite service placement algorithm and experimental results have been suppressed in the public version of this report as this content is currently under review for publication in a scientific journal.

2.3 On-demand Service Placement

2.3.1 Motivation

In a FUSION architecture, it is impossible/inefficient to pre-deploy instances of all services in all execution zones, especially for very sensitive (BW/RTT latency) and/or long-tail services for which it is impossible to pre-deploy (the right amount of) services in these remote nodes.

Specifically, it is reasonable to assume that the average size of an execution zone is proportional to the distance to the end users: the closer to the user, the smaller the execution zone or data centre. Moreover, the number of execution zones also drastically increases as you reduce the average and/or maximum distance between any user an its closest execution zone. Combined, this means that you drastically increase fragmentation of resources compared to a single huge centralized data centre that is reachable by everybody.

Consequently, except perhaps for the top *X* most popular services, it is infeasible and not cost-effective to pre-deploy a number of instances with corresponding session slots in all these distributed (and possible small) execution zones. For the long tail services, it may even be impossible to predict with high enough accuracy what the demand would be in a particular region at a particular moment in time.

As such, it is preferable to go for a more on-demand driven approach in such cases, where instances are only (fully) deployed in some smartly-chosen location when requested/needed. However, to efficiently do this, we need fast provisioning and deployment strategies.

2.3.2 On-demand Service Provisioning

On-demand provisioning is an interesting method to avoid cluttering edge nodes with service instances which are rarely used. Rather than pre-deploying instances that are mostly idle, we aim to store components nearby so that, upon request arrival, an instance can boot up and respond to that request in the desired response time. Docker [Fi14, Me14] could be used to facilitate on-demand provisioning and has seen an increase in popularity. Docker is a popular technology to build services as a suite of small components with a very specific functionality which can be deployed separately.

Docker uses operating-system-level virtualization for the deployment of applications inside lightweight software containers. Docker services consist of a layered file system (layers) where each layer can be reused by other services and stored on different locations. Software containers share the same operating system kernel as the host system, allowing Docker to deploy a running service instance almost instantly.

With multiple services reusing the same layers, the challenge is to find the optimal location for each layer to allow Docker to retrieve layers and deploy an instance quickly so the service can respond to the request within the desired response time. Research on optimal service placement typically minimizes the cost to uphold a certain Service Level Agreement (SLA) based on expected demand. However, very little research goes towards the **long-tail services** with no predictable demand patterns.

We speak of long-tail services if the average demand per time unit (Erlang) is less than one for users in a nearby geographical area.

FUSION therefore proposes a service placement algorithm which maximizes the amount of clients we can serve on-demand. Unlike most existing placement algorithms, we focus on long-tail services and consider that clients may connect from any given location to request a service located in the network. Using the layered file system introduced by Docker, our algorithm places the layers on nearby storages so that users can connect to a server, download the service layers and deploy a service instance within a desired response time. Docker handles the layer retrieval and instantiation so we only concern ourselves with finding the optimal location for the layers (Figure 11).

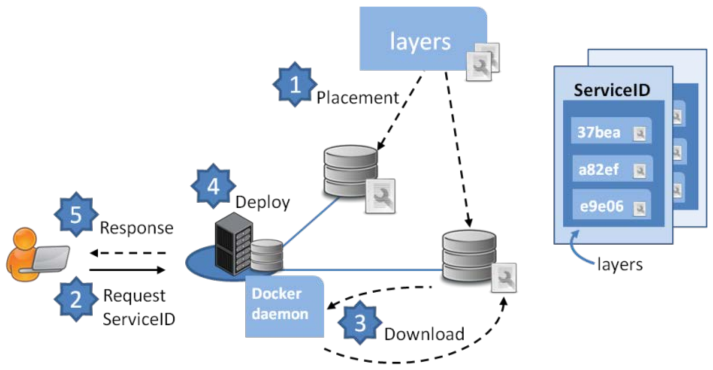


Figure 11 – Our algorithm places Docker image layers on remote storages so that, upon request arrival, Docker retrieves the required service layers, deploys an instance almost instantly and the instance responds to the client, all within the desired response time.

The on-demand service placement algorithm and experimental results have been suppressed in the public version of this report as this content is currently under review for publication in a scientific journal.

3. HETEROGENEOUS DISTRIBUTED CLOUD PLATFORM

In this Chapter, we motivate the need for a heterogeneous (distributed) cloud platform to be able to cost-efficiently and predictably manage the typical FUSION application services: demanding interactive real-time (media) services running in a cloud environment, and discuss the benefits of using a lightweight composite application service mode. We then present and discuss a number of key monitoring, profiling and visualization techniques for dealing with this heterogeneity, both from an application as well as runtime perspective. We apply this onto two key FUSION orchestration concepts, namely session slots and evaluator services, and analyze their impact in detail. We also evaluate various inter-service communication mechanisms as well as present the results from an extreme service density study. We conclude with the presentation of a heterogeneous cloud platform design that could incorporate all these features in a dynamic and self-learning manner.

3.1 Motivation

The key motivation for a heterogeneous (distributed) cloud platform has already been discussed at great length Deliverable D3.2, particularly in Section 2.2, where we focussed on the key challenges and opportunities of a heterogeneous cloud environment for managing demanding workloads, services and network functions.

In summary, our main vision is that in the future, more and more demanding workloads and bearer plane services will also be deployed in the cloud, as depicted in Figure 12.

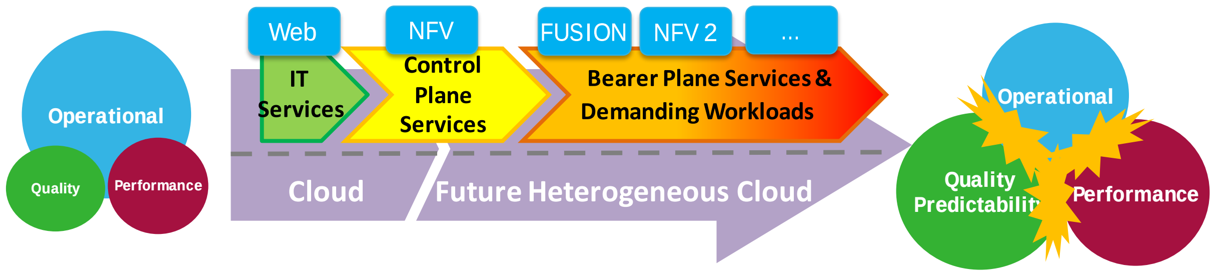


Figure 12 – The Future of Cloud.

However, as these demanding and sensitive real-time applications are currently typically deployed on dedicated specialized hardware, deploying them as a cloudified service on an unknown general-purpose (third-party) cloud environment can result in substantial performance and QoS drops, with the net effect that the operational benefits of deploying on a general-purpose cloud with standard cloud nodes are counterfeited by the decrease in efficiency, resulting in the end in a higher TCO.

As such, the FUSION vision is that for such applications, network functions and workloads to be cost-efficient in a cloud environment, dynamic and adaptive heterogeneous cloud environments are necessary, taking into account the application requirements and the HW/SW platform characteristics, and automatically tuning the software platform and hardware infrastructure based on these requirements and the underlying capabilities and limitations. Apart from being distributed, these heterogeneous cloud environments will consist of novel hardware infrastructures such as micro-servers that are designed for energy efficiency, as well as hardware accelerators for regaining some of the performance efficiency losses from using general purpose hardware only.

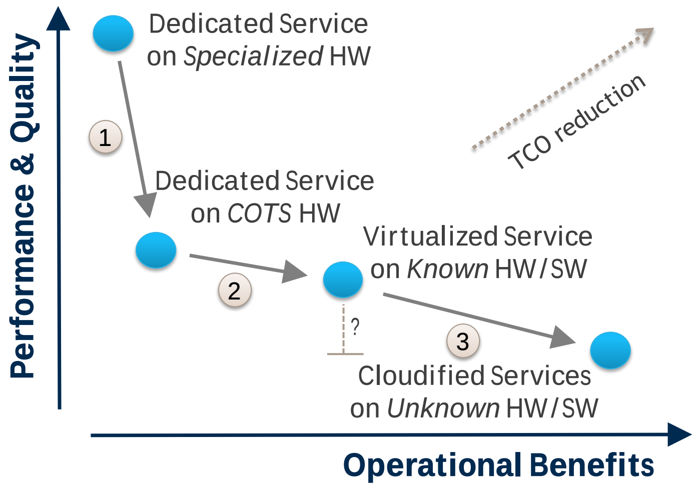

Performance & Quality
Dedicated Service on *Specialized* HW
1
Dedicated Service on *COTS* HW
2
Virtualized Service on *Known* HW/SW
?
3
Cloudified Services on *Unknown* HW/SW
TCO reduction
Operational Benefits

Figure 13 – Trade-off between operational benefits & efficiency when virtualizing and cloudifying demanding services on unknown COTS hardware and software infrastructures.

This already starts to become visible in the industry, where big players like Google and Facebook and Amazon, both owning an impressive array of powerful data centres, are taking various initiatives for increasing the overall efficiency of their data centres. For example, Facebook recently announced to be working in next generation open hardware specifications as part of their Open Compute Project (OCP) initiative [Mi16]. These servers will consist of a heterogeneous mix of SSDs, GPUs, NVM and JBOFs to accelerate their infrastructure for better dealing with the new and more demanding applications, including artificial intelligence and virtual reality. Additionally, there are working together with Intel for designing a new processor and corresponding server infrastructure that is optimized for these new workloads.

Similarly, Google recently joined the same Open Compute Project, contributing their new 48V rack specification and a new form factor to allow OCP racks to fit into existing data centres [Go16]. These new specifications allow for a 30% better energy efficiency, and more importantly, also are more cost effective in supporting new higher-performance systems, incorporating high-performance CPUs and GPUs.

On the more distributed nature of clouds, one can clearly observe the massive new investments that Google and Amazon are making and have recently been making in deploying additional huge cloud data centres around the globe. Amazon currently already has 12 data centres across the globe (of which two in Europe now), and Google, who currently only has 4 for its public Google Cloud Engine (GCE), recently announced to be investing in installing 12 additional data centres across the globe. In their announcement [Go16b, Su16], they stated that they are '*opening up these new regions to help Cloud Platform customers deploy services and applications nearer to their own customers, for lower latency and greater responsiveness. With these new regions, even more applications become candidates to run on Cloud Platform*'.

This chapter builds upon that vision of a future distributed and heterogeneous cloud, and investigates various techniques for efficiently and dynamically dealing with this amount of heterogeneity, both from an application perspective as well as cloud infrastructure perspective. More background on the various challenges and opportunities we see, can be found in Sections 2.2.2 and 2.2.3 of Deliverable D3.2, respectively.

3.2 Lightweight Composite Application Service Model

3.2.1 Motivation

The distributed and heterogeneous nature of a FUSION execution platform demands for a more lightweight and composite application service model. By lightweight, we mean that the services or service components need to be relatively small in size and fast to deploy, so that in a distributed cloud environment, service components can be downloaded, provisioned and instantiated within seconds or even sub-second, something which is clearly not feasible with fat VMs containing complex services that take (tens of) minutes to fully provision and instantiate.

By composite we mean that a service should be decomposed into a smaller number of service components, so that they are more lightweight, can be deployed across distributed cloud nodes (but only when this is beneficial for the application), and last but not least, so that each service component can be deployed more cost-efficiently in its most optimal heterogeneous runtime environment. A key enabler for this is efficient inter-service communication mechanisms, so that the benefits of a distributed heterogeneous composite service are not nullified by the overhead of inter-service communication.

In this section, we will further elaborate on various models and technologies for implementing such lightweight composite service model.

3.2.2 Microservice Architecture

An increasingly popular software architecture style is a microservice architecture, where complex applications are broken down and decomposed in a number of relatively loosely-coupled services that each provide a single function that can be accessed and manipulated via a specific (and typically language-independent) API. Some of the key properties of such software architecture are that each service (component) can have its own lifecycle (development, testing, deployment, etc.), can be implemented in its own programming language and/or framework, and as such can also be easily replaced by another service that implements the same functionality. This allows for a more *devops* style of working, where small teams can focus on the entire process of that single service (component) in a continuous software delivery process, where new versions of one component can be released independently from the other components. In this more symmetrical model, each service component acts as a producer/server for other components and as a consumer/client to other components.

The special purpose single-function nature of these microservice components in a heterogeneous cloud have the additional advantage that they can be mapped more easily on their respective optimal runtime environment, aligning particularly well with special-purpose cloud nodes and micro-server architectures, where cloud nodes either have specific hardware capabilities, or are broken down into smaller more modular hardware components that can be dynamically reassembled based on the application requirements. For example, a real-time media service typically consists of a number of decoding functional blocks, rendering blocks and encoding and streaming blocks. Although all these functional blocks could easily be running on the same general purpose cloud hardware platform, each of these individual functional blocks could run much more cost-efficiently in case they can be deployed on more specialized hardware nodes (e.g., GPU-enabled node for rendering, real-time encode/decode nodes, etc.), taking into account the specific runtime requirements of each functional block.

Lastly, by splitting the application into various microservices, not only can they be deployed more efficiently, their loosely coupled nature also allows each microservice to be easily reused across multiple independent applications. For example, a decode, encode or render microservice could easily be reused for different types of applications, increasing efficiently and reducing deployment time.

A key issue with microservice architectures however is the importance of correctly decomposing an application in relevant and loosely coupled service components, and the importance of carefully designing the best API for each component. If the decomposition is done in a wrong way or the APIs are designed in a wrong way, then many of the benefits of a microservice architecture can very quickly turn against itself. Two key aspects to take into account when (de)composing an application are the reusability of each component and how loosely coupled each component is from the other. In case of real-time media services, inter-service communication is also a crucial aspect to consider to avoid wasting many (compute/network) resources when applying a microservice model onto such application.

3.2.3 Multi Session Slot Service Architecture

To increase reusability and efficiency of these microservices, each microservice component should be able to serve a particular number of requests from different clients independently in parallel. In this way, many static and dynamic resources can be shared both in space and time, reducing the need for deploying separate and isolated instances of the same functionality. In FUSION, we propose a multi-session slot service architecture to specifically take into account the discrete amount of resources each (long duration) session consumes, and how many resources are still available. This can be enabled for each individual microservice component, and it is FUSION that takes care of the complexity of balancing the requests across all instances. As such, this architecture improves the lightweightness of the service deployment, as there is not a 1:1 mapping between an instance and a session.

3.2.4 Implementations

Such lightweight composite application service model can be implemented in cloud environment in a number of ways. In this section, we discuss two key models, namely a service container model and a unikernel model.

3.2.4.1 Lightweight Containers

In the last few years, lightweight (Linux) containers have become increasingly popular as a new way of packaging, provisioning and deploying lightweight applications, as an alternative to the classical full virtual machines, which are much more heavyweight as they typically wrap an entire OS environment along with the application and all its services. This was to a large extent the merit of Docker and specifically due to the APIs, tools and ecosystem that they built around it for packaging, provisioning and deploying applications.

Lightweight containers were already discussed in detail in D3.2, so we will not repeat in details the key properties and advantages of containers. In summary, lightweight containers is an OS-level isolation mechanism provided by the kernel for isolating applications (instead of full machines in case of VMs) while sharing the same OS. Due to the lightweight nature, it can provide this isolation with very little overhead, and enables to start/stop applications wrapped in such containers within less than a second. Due to the lack of an additional virtualization layer and OS, it also has much better runtime characteristics than VMs, which is especially interesting for demanding applications.

Lightweight containers in general and Docker containers in particular efficiently enable a microservice-enabled software architecture, supporting also the lightweight composite service model we are seeking. This was already illustrated in previous Deliverables (i.e., D3.2 and D5.2), and is reflected as well through additional experiments in this deliverable and D5.3.

3.2.4.2 Unikernels

Lightweight containers however are not the only mechanism for implementing a microservice architecture in general or our lightweight composite application model in particular. Although containers have the advantage of being lightweight with almost zero overhead and start-up time, their key disadvantages are still the rather weak isolation due to the shared kernel and OS, the shared OS (in terms of version, features and performance settings), and live migratability. Classical virtual machines on the other hand provide much stronger isolation between VMs/apps, provide a fully contained and customized OS and software environment, live migratability, but at the expense of being heavyweight, with typically non-negligible OS and resource overhead and start-up time.

An upcoming technology that tries to combine the best of both works however are unikernels. In short, a unikernel is a lightweight (virtual) machine, typically (but not exclusively) running on a standard hypervisor, where the OS is typically directly integrated into the application. Booting a unikernel in fact simply involves booting the application along with some minimal hardware initialisations. As such, it has the same lightweight benefits and agility of containers in terms of image size and start-up time, but combined with the stronger isolation and OS customization/optimization benefits of VMs.

Some key disadvantages of unikernels on the other hand are the need for a custom tool chain or specific runtime environment, limited debugging capabilities in case something goes wrong, etc. Target areas for deploying unikernels are cloud applications and IoT, where these lightweight unikernel applications could be directly deployed on the IoT device with minimal overhead. Typical unikernel properties include the use of a single address space (no separate kernel and application space), application and cloud tuned pluggable OS library components, etc.

An overview of the application stacks involved for various virtualization and isolation mechanisms is depicted in Figure 14, and clearly shows the various layers involved when using a particular technology. In the unikernel approach, there is no notion of a (separate) OS and runtime. Instead, everything is integrated into a single bootable application and tuned for the particular workload. For example, in case of a I/O intensive application, application-specific OS components could be included that are tuned for that specific application, possibly resulting in significantly higher throughputs compared to using standard OSes (with or without virtualization and/or containers).

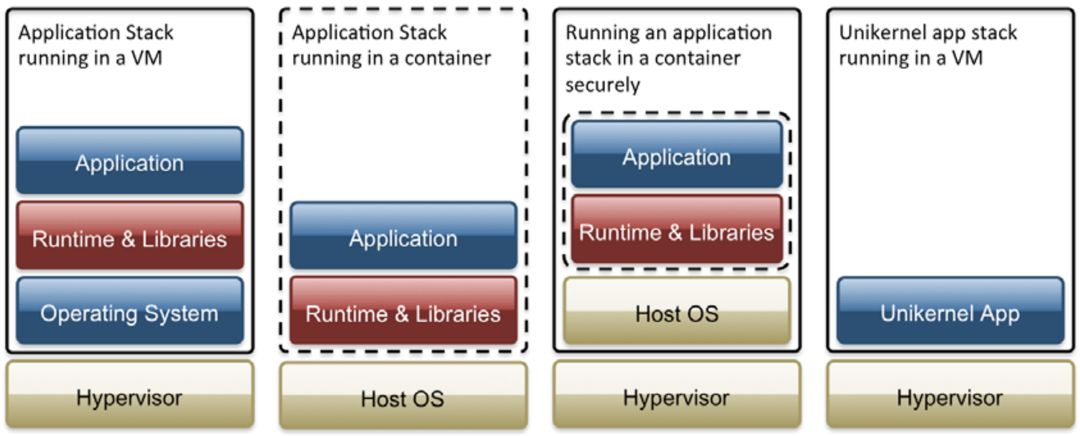


Figure 14 – Different application virtualization and isolation mechanisms (Image credit: Xen Project).

As these unikernels are so application-specific, different types and implementations of unikernel frameworks exist, each focussed or tailored for a particular use case or runtime framework. An overview of these technologies and frameworks can be found on http://unikernel.org. A list of the key unikernel technologies:

- ClickOS: a click-based unikernel OS, using the Click router as foundation
- Clive: targeting distributed HPC cloud environments, based on GO and inspired by Plan9
- Drawbridge: providing lightweight virtualization in Windows, using a library OS
- HaLVM: a Haskell compiler to run Haskell applications directly on top of a Xen hypervisor
- IncludeOS: a library OS for cloud services
- LING: transforms Erlang code into LING unikernels

- MirageOS: a clean-slate library OS for building secure, high-performance network applications in cloud and mobile
- **Rumprun**: allows existing POSIX software to run as unikernel, using kernel component drivers
- **OSv**: a new OS specifically designed for cloud VMs, supporting C, JVM, Ruby and Node.js application stacks
- Runtime.js: OS library for the cloud running JavaScript (node.js style) on KVM

Within the FUSION prototype, we experimented with two unikernel frameworks, namely Rumprun and OSv.

Rumprun allows to run unmodified POSIX applications as unikernels, either on raw hardware (e.g., in embedded systems or in the IoT space), or on hypervisors such as Xen as well as Qemu and KVM). To build a Rumprun unikernel, one needs to cross-compile an application using their custom tool chain and subsequently "bake" the resulting binary into a unikernel, during which one can select the relevant kernel bits that are needed. For this, they use NetBSD kernel modules. The baking process is done for a particular hardware environment (e.g., xen, hw, hw_virtio, etc.), and particular kernel components can be included, optimized for the application and/or hardware environment. Running a Rumprun kernel can be done either via their rumprun wrapper or directly using Qemu/Xen, or by writing the bin-file onto a physical disk.

OSv on the other hand works in a very different manner. OSv uses as a basis a library OS that was built from scratch, optimized for the cloud, and possibly optimized for the application needs. It runs on a range of modern hypervisors (e.g., KVM, VirtualBox, VMWare, GCE, etc.). Compared to Rumprun, one can use existing tool chains for building the application. However, the application needs to be built as a shared library. OSv provides a similar look-and-feel for building, provisioning and running OSv unikernels as the Docker ecosystem. To this end, OSv provides a capstan tool similar to the Docker client application, as well as provides a (shared) repository for managing and exchanging unikernel images. It has a manifest file (either Makefile or Dockerfile in style) with instructions for building and running an OSv application, and also used the concept of parent images to recycle and specialize general-purpose unikernel images into specialized versions (e.g., providing additional shared libraries).

Note also that Docker itself recently acquired a unikernel solution and is currently in the process of integrating this as a backend into their Docker platform, enabling the same Docker APIs and ecosystem to be leveraged for managing applications as unikernels apart from containers.

We experimented with early versions of both Rumprun and OSv within the FUSION prototype. Specifically, we recompiled some of the prototype application service components such as the EPG service and decoder service as OSv and Rumprun unikernels and deployed them as KVM VMs. W.r.t. image sizes, our FUSION multi-session slot video decode service is about 70 MiB in size as a Rumprun unikernel compared to about 90 MiB as an OSv unikernel, though the OSv unikernel does support multiple cores, whereas Rumprun does not (yet). The EPG core service on the other hand is about 20 MiB size as a Rumprun unikernel and 30 MiB as an OSv unikernel. The start-up time of the service was below 1 second (even though a virtual machine was created). Some issues we encountered were the difficulty of debugging in case there is an application crash (in which case the entire VM simple stops), the support of only a single core machine in case of Rumprun (which conflicts with our multi-session slots model), and the networking hiccups and stuttering that were observed for OSv when streaming decoded raw frames over TCP sockets on the virtual network interface.

Nevertheless, unikernels appear to be a promising alternative solution to Linux containers for implementing the lightweight composite FUSION service model, providing stronger isolation and integration in existing cloud platforms, combined with inherently fast boot times (though a non-optimized existing unikernel-agnostic cloud orchestration platform may introduce much higher boot times).

3.3 Profiling

3.3.1 Motivation

In FUSION, we target demanding real-time services to be deployed on a distributed heterogeneous cloud environment. This means that the application services typically will have very specific resource and/or runtime requirements to be able to cost-efficiently run in a particular runtime environment. On the other hand, the heterogeneous cloud environment provides very specific and tailored resource and runtime capabilities to application deployed in such environments. Example resource capabilities include specific hardware accelerators (e.g., GPUs, video encoders or decoders, fast memory and storage, etc.); example runtime capabilities include real-time guarantees and strong performance isolation when deployed on a particular environment, low-cost or highly energy efficient runtime environments, etc.

In general, the performance of a service on a runtime platform is influenced by numerous factors, including the following ones:

- Hardware capabilities of platform (hardware acceleration)
- Multi-tenancy
- Virtualization technique (BareMetal, VMs, hypervisor, containers, ...)
- NUMA and pinning
- Scheduling and real-timeness
- Clock governance
- ...etc.

This diverse pool of resource consumers (i.e., the applications) and producers (i.e., the cloud environments) need to matched with each other: application providers want to deploy their applications on the most cost-efficient runtime environment that provides sufficient QoS, whereas the cloud providers want their environment to be used as optimal as possible: higher application densities while providing sufficient QoE results in higher revenues. Given the multitude of influencing factors, a general approach to perform an estimation of service capacity is quite challenging.

As such, there is a clear need from both sides for profiling: application providers want to quickly find the optimal runtime environments from the wide range of available (a priori unknown) runtime environments, whereas cloud providers want to ensure the provided runtime environments provide the expected runtime characteristics, and also may want to investigate how to improve the available runtime environments to more efficiently deploy particular applications onto its cloud infrastructure.

Consequently, in all profiling mechanisms and tools described in this section, it is important to keep in mind that there is always this duality that these tools can be used by both parties, but with different intentions.

In the following sections, we first study and present a number of profiling methods. Afterwards, we discuss the concept of evaluator services in detail, linking the results of the various profiling methods presented.

3.3.2 Application & System Profiling

3.3.2.1 Motivation

We want to efficiently characterize (i) how a particular application behaves on particular HW/SW runtime environments, and vice versa (ii) how a particular runtime environment will behave for particular applications. Instead of trying out all possible combinations, one can also try to find (i) applications that behave similar on a range of runtime environments, and, vice versa (ii) runtime environments that behave similar for a range of applications.

This can be used both as input for the evaluator services, to see whether one can reduce the number of evaluations, by using general purpose profiling applications instead of special application-specific profilers. For the heterogeneous cloud platform provider, this would provide valuable feedback on learning what environment types work well or are less effective for particular application services, again without having to try all possible combinations.

In this section, we present our results of an initial study towards this objective, focusing initially on CPU-intensive benchmarks, and specifically the throughput characteristics. The real-timeness aspect is briefly covered in Section 3.3.2.5.

3.3.2.2 Methodology

In this section, we first present the various benchmarks and test platforms we used for this study.

3.3.2.2.1 Benchmarks

We used two types of benchmarks for this experiment:

- Spec CPU 2006 benchmark suite, consisting of integer and floating point CPU-intensive benchmarks, typically used for assessing the performance of a particular hardware system: *astar, bwaves, bzip2, cactusADM, calculix, dealII, gamess, gcc, GemsFDTD, gobmk, gromacs, h264ref, hmmer, lbm, leslie3d, libquantum, mcf, milc, namd, omnetpp, perlbench, povray, sjeng, soplex, sphinx3, tonto, xalancbmk, zeusmp.* More information can be found on the SPEC website: http://spec.org/.
- Our custom FUSION media benchmarks, developed in our Vampire framework, representing various media processing kernels: imagecopy, YUVconversion, alphablending, resampling, decode, encode, transcode, background extraction.

The SPEC benchmark suite calculates a "SPEC" score, which is a rate measure of how much faster the measured hardware system is w.r.t. some reference base hardware platform (an old Sun Sparc machine). This rate score is measure as follows:

$$rate = \#instances * reference\_elapsed\_time / measured\_elapsed\_time$$

We use this rate score for both benchmark suites as basis for doing the subsequent correlation analysis.

3.3.2.2.2 Test Platforms

These benchmarks have been deployed on a wide range of low-power as well as high-performance hardware platform. Note that in this study, we deployed each benchmark bare metal; consequently, other aspects such as virtualization overhead, noisy neighbors, etc. w.r.t. application and/or system profiling are not taken into account in this study. Below an overview of the various hardware platforms. We gave each platform a name and a corresponding short name to be used in the various figures during the discussion.

- Hercules (he): AMD Opteron 6174, 2x12 cores @ 2.2 GHz
- Mozes (mo): Intel Core i7 975 4-core+HT @ 3.3 GHz
- vwall-node0 (vw): Intel Xeon E3-1220 4-core @ 3.1 GHz
- xeonv2 (xe): Intel Xeon E5-2690v2 10-core @ 3.0 GHz
- sbp-node1-6 (sb): Inten Xeon E5-2680v3 @ 2.5 GHz
- Silver (si): Intel Avoton C2750 8-core @ 2.4 GHz

- Nuc (nu): Intel Core i3 5010U, 2-core+HT @ 2.1 GHz
- STCK1 (st): Intel Atom compute stick STCK1A8LFC

3.3.2.3 Principal Components Analysis

We ran each benchmark in a particular multitude on all hardware platforms. To be able to correlate either the benchmarks or the machines, we used Principal Component Analysis (PCA), which is a standard statistical technique for transforming possibly correlated variables into a set of linearly uncorrelated variables. These uncorrelated variables are also called the *principle components*.

As the transformation is done in such a way that the first principle component will contain the largest variance to be extracted from the sampled data, and so on, typically after transformation, only the first few principle components are considered to be valuable (the others have almost no variation an thus are typically of little importance).

A typical example of this orthogonal transformation is depicted in Figure 15.

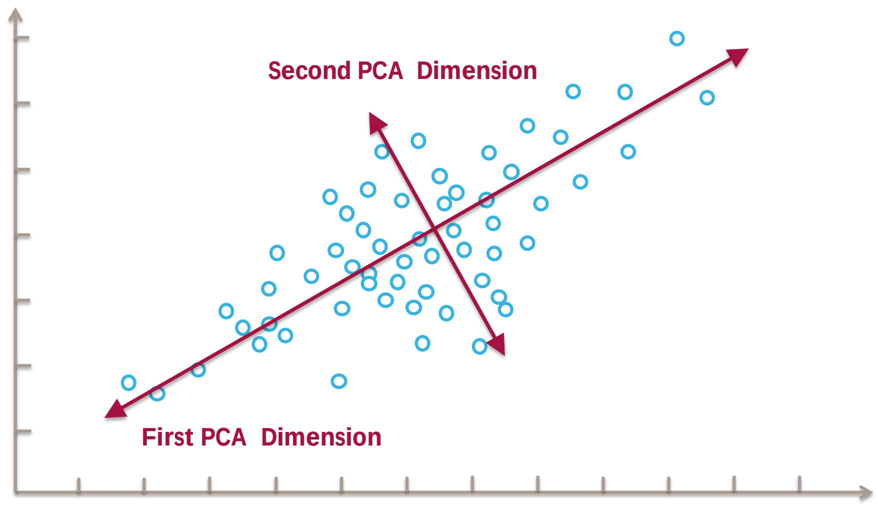


Figure 15 – Visual representation of PCA transformation.

In this example, the two-dimensional data is transformed along the principle components. As mentioned earlier, in many cases, PCA is also used to map highly-dimensional correlated data into low-dimensional linearly uncorrelated data.

This statistical method is not only used to better understand the typically correlated input data, it can also be used for making prediction models. In our study, we used the *prcomp* method of the statistical software package *R* for generating the principle components and corresponding main graphs.

3.3.2.4 Evaluation

3.3.2.4.1 Initial example

In the first part of this analysis, we will first try to classify the different machines based on the various benchmarks listed above. We focus on the FUSION media benchmarks first, and focus on classifying different flavours of the same vwall machine, using only one CPU core up to all 4 available CPU cores. In the graphs, this is represented as either vwall-node.*x*, or vw*x*, where *x* represents the number of available cores. This roughly corresponds to various HW flavours when running an application in the cloud.

The results of this PCA transformation is shown in Figure 16.

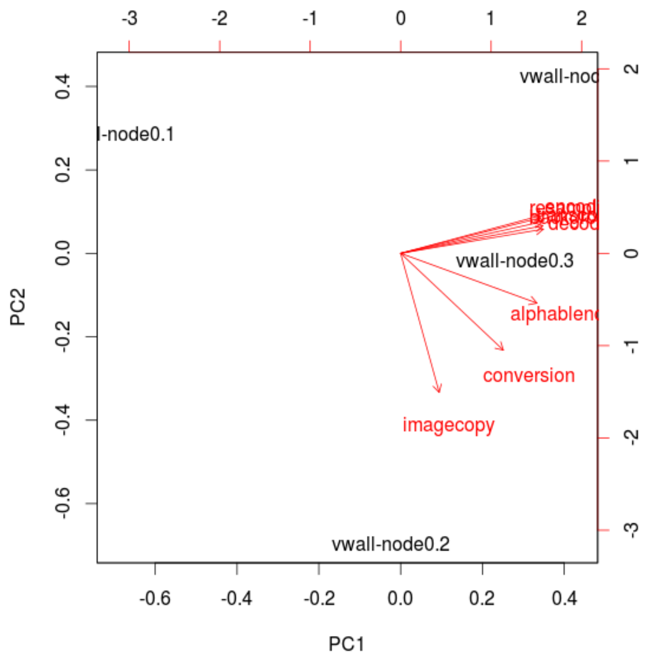


Figure 16 – PCA biplot of the vwall nodes using the FUSION benchmarks.

The various vwall HW nodes represent the different data samples in the newly transformed space. The red arrows and labels represent the respective contributions of each original dimension w.r.t. the transformed PCA dimensions.

From this, a number of observations can be made. Firstly, all red arrows point in the positive direction of PC1 (i.e., they point right), meaning they all have a positive contribution w.r.t. the first principle component (PC). All benchmarks except conversion and imagecopy have the largest positive contribution. This means that machines with a high value for this first PC will have a high rate for these benchmarks. Note that these benchmarks are the most CPU-intensive benchmarks, whereas imagecopy and conversion are ore memory-bound (but still CPU intensive). Looking at the vwall-nodes flavours, one can also clearly see that they are ordered along the PC1 axis according to the number of active cores. Combined, one can conclude that the first PC roughly represents the absolute CPU throughput, which also happens to contain the largest amount of variation: deploying these CPU-intensive benchmarks on 4 cores will result in a much higher throughput than deploying them on only a single core. Note that for less-CPU intensive benchmarks, this variation would be much less dominant, and as such their contribution would be much less. This is observed partially through the conversion and imagecopy benchmarks, for which adding more cores does not result in a proportional increase in rate, as the memory/caching bottlenecks start to dominate instead.

This aspect is actually reflected in the second (but less dominant) principle component, where you see a much large contribution of these benchmarks along the vertical axis. The fact that they have a negative contribution is not relevant here, as one could easily make another transformation that flips the direction of the second PC.

When looking at the vwall nodes, one can observe that a vwall node with 2 cores activated, performs “best” along the second PC (i.e., they have the lowest PC2 value), whereas the vwall node with 3 and 4 cores activated seem to perform increasily worse along this principle component. Knowing that this PC represents how well these machine nodes perform for the memory-intensive applications imagecopy and conversion, one can conclude that the second PC represents memory-bandwidth efficiency: when increasing the available cores from 1 to 2, there is a relative boost in throughout efficiency. Further increasing the number of cores to 3 and 4 still increases absolute performance on these nodes (see PC1 projections), but not as efficiently as in case of 2 cores: many CPU cycles are wasted for memory-intensive applications waiting for the caches and/or system RAM. The CPU-intensive applications on the other hand have barely any contribution (or even a negative contribution) along this second PC, so they do not suffer from this bottleneck.

Note that we only show the first two PCA dimensions (as often is done), as these explain over 95% of all data variance, as depicted in Figure 17.

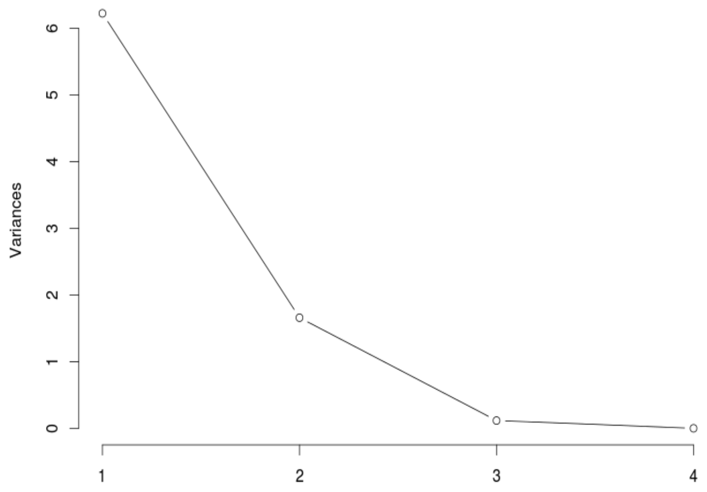


Figure 17 – Variance of principle components for the vwall nodes with the FUSION benchmarks.

The (factor) contributions of each benchmark for the first two principle components are also depicted in Figure 18. This shows the same data as the red arrows, but presented as points rather than arrows. From this, for the vwall machines, one can also conclude that the decode, resampling, encode and backgroundextraction benchmarks have little discriminating power for classifying these vwall node flavours: one could use either one, instead of using all of them.

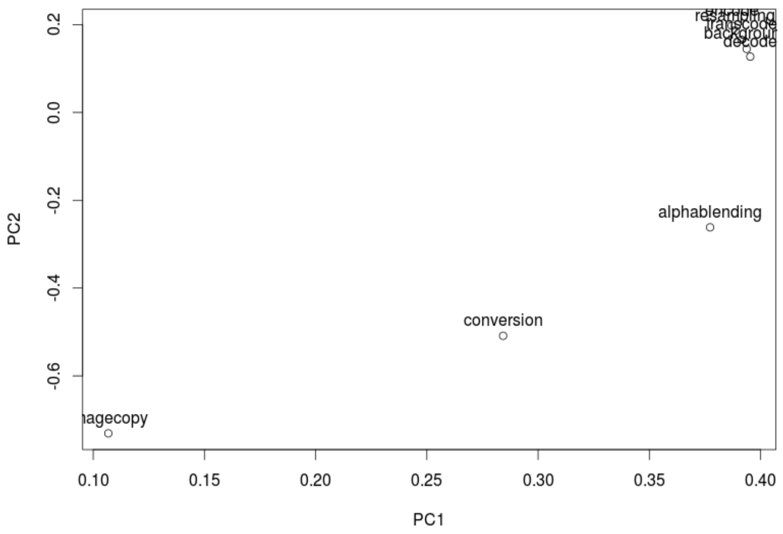


Figure 18 – Factor contributions of each benchmark for the first two principle components for the vwall nodes with the FUSION benchmarks.

3.3.2.4.2 Classifying Machines

Now that we have explained how to interpret the results from the PCA analysis for this use case, we can now apply the analysis on a larger data set, including more machines and/or benchmarks. In Figure 19, we show the transformed machine data along the first two principle components.

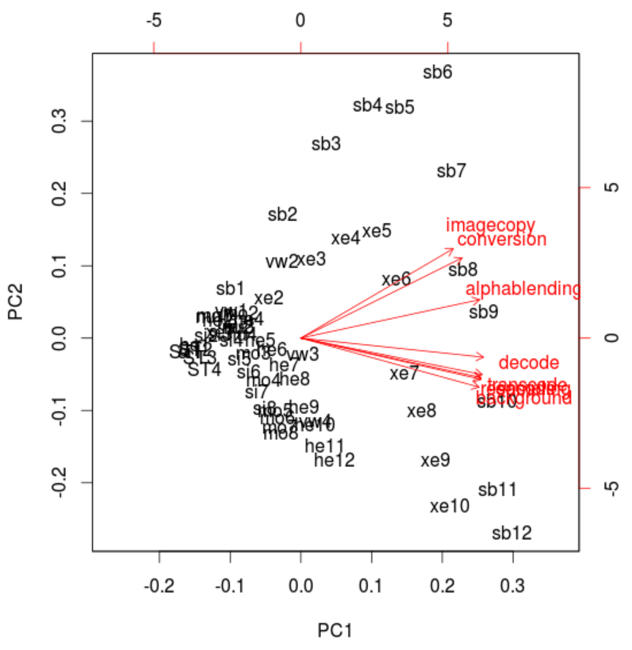


Figure 19 – PCA biplot of all machine nodes using the FUSION benchmarks.

Interestingly, similar observations and conclusions can be made as in case of the vwall nodes only: all benchmarks have a positive contribution w.r.t. the first PC, and this typically also lines up with an increasing number of used cores (see for example xe*, sb*). So the first PC, which contains the largest amount of variation, reflects roughly the raw CPU throughput capabilities. Notice also that the modern high-performance servers also perform much better than the older server (he*) or the lightweight cores (ST*, si*).

When looking at the second PC, one can see again that the imagecopy and conversion benchmarks have the highest contributions (see also the factor contributions graph in Figure 20 below). When focusing on the sb* and xe* nodes, one can again see the same parabolic pattern: it first increases, slowly reaches an optimum, and then drastically decreases again. In both cases, this optimum is reached when half of the cores are being used, meaning this is the sweet spot for running these memory-intensive applications: using more cores will still result in higher throughput rates, but with increasingly lower increments. For a cloud platform provider, in case an application is profiled as being memory-intensive, this means it may be better deploying a non-memory intensive application on the remaining CPU cores, rather than stuffing the CPU with other memory-intensive applications, as this will only cause interference and noisy neighbours, resulting in reduced efficiency.

In Figure 20, the factor loadings are shown again, but now applied to the full data set of all machines. A similar pattern can be observed as in Figure 18: there is a relatively large discriminating behaviour between the imagecopy, conversion and alphablending benchmarks, but less between the other benchmarks (though slightly more that in the small data set).

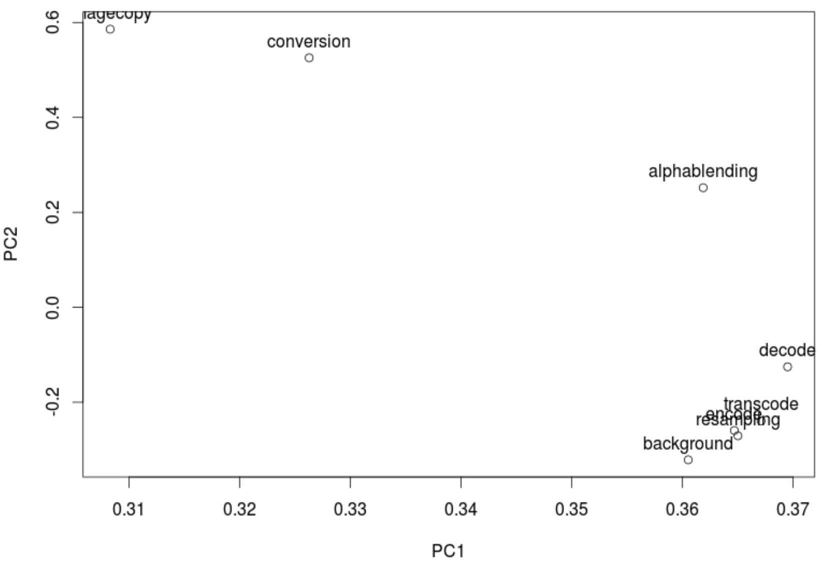


Figure 20 – Factor contributions of each benchmark for the first two principle components for all machine nodes with the FUSION benchmarks.

For completeness, we also show the amount of variation captured in each PC in Figure 21. Clearly, the first PC contains most of the variance for these CPU-intensive applications, where varying the number of available CPU cores has a dramatic impact on the resulting performance rates. Note also that, as we use the spec rates for our data, it is clear that only performance-impacting aspects and differences can be measured in this analysis. In case other non-performance related values would be included (e.g. power, cost, etc), other conclusions could be drawn when doing these analyses.

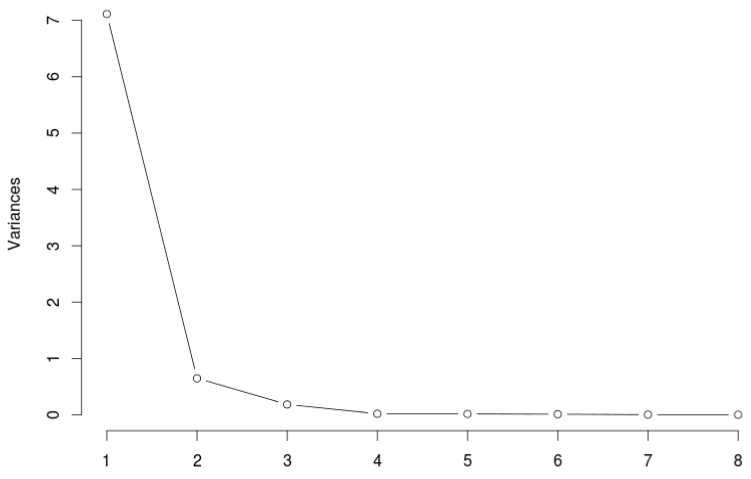


Figure 21 – Variance of principle components for all machine nodes with the FUSION benchmarks.

Next, we now also add the SPEC 2006 benchmarks. The resulting transformed data is presented in Figure 22. Interestingly, the graph again looks very similar.

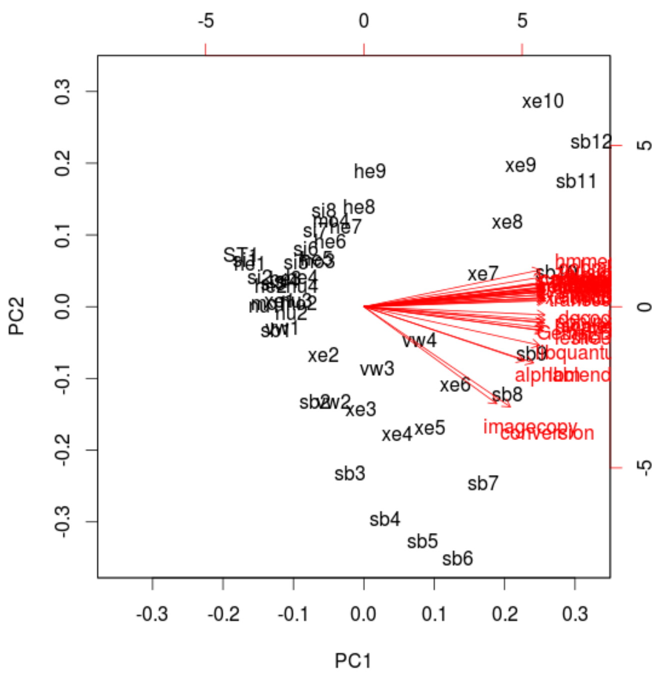


Figure 22 – PCA biplot of all machine nodes using the FUSION and SPEC2006 benchmarks.

When looking at the individual factor contributions of each benchmark as shown in Figure 23, one can clearly see that most SPEC2006 benchmarks have very little discriminating power w.r.t. each other, at least not for classifying the various machine nodes: most will return roughly a similar relative SPEC rate on the various machine nodes. This knowledge can help when improving cost-efficiency of developing and deploying evaluator services, or when profiling cloud platforms using benchmarks.

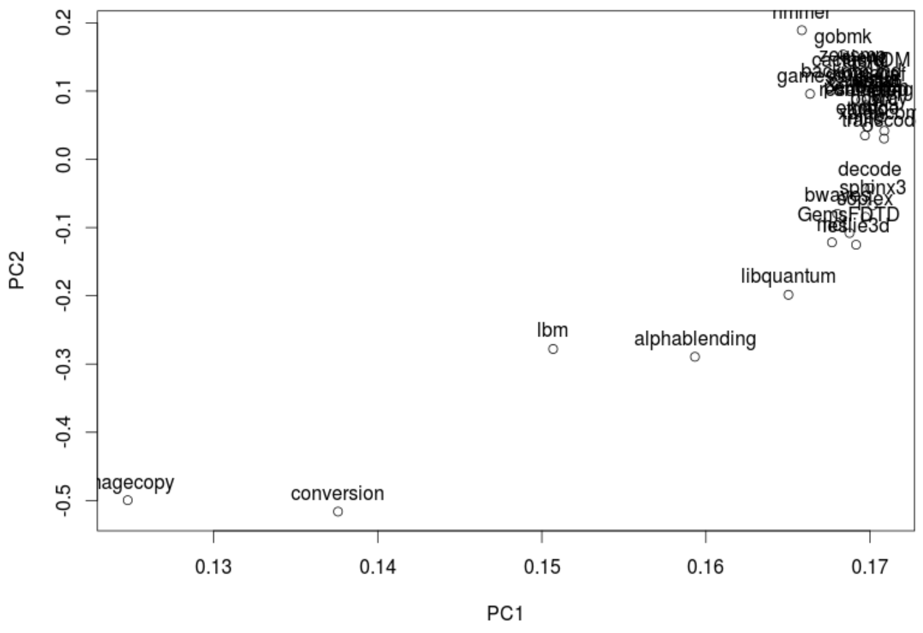


Figure 23 – Factor contributions of each benchmark for the first two principle components for all machine nodes with the FUSION and SPEC2006 benchmarks.

Again, for completeness, we also show the amount of variation per PC. Obviously, due to the abundance of CPU-intensive applications all measuring roughly the same thing, the first PC outweighs any other PCs. Rerunning the analysis with a clustered subset of benchmarks would result in a better representation.

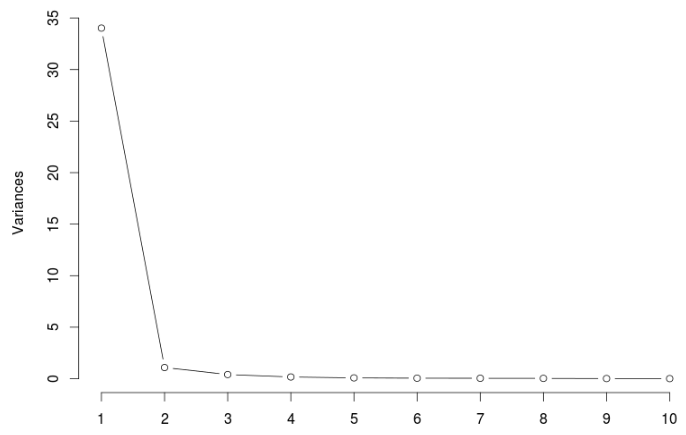


Figure 24 – Variance of principle components for all machine nodes with the FUSION and SPEC2006 benchmarks.

3.3.2.4.3 Classifying Benchmarks

In the previous section, we use PCA analysis to classify machines based on the rate results from different benchmarks. We also already observed various similarities between benchmarks. In this section, we now use the same PCA analysis for explicitly classifying benchmarks. For this, we simply transpose the data and rerun the PCA analysis.

Again, we start with a subset of the data, using only the various flavours of the vwall node for classifying the FUSION media benchmarks. The results are depicted in Figure 25.

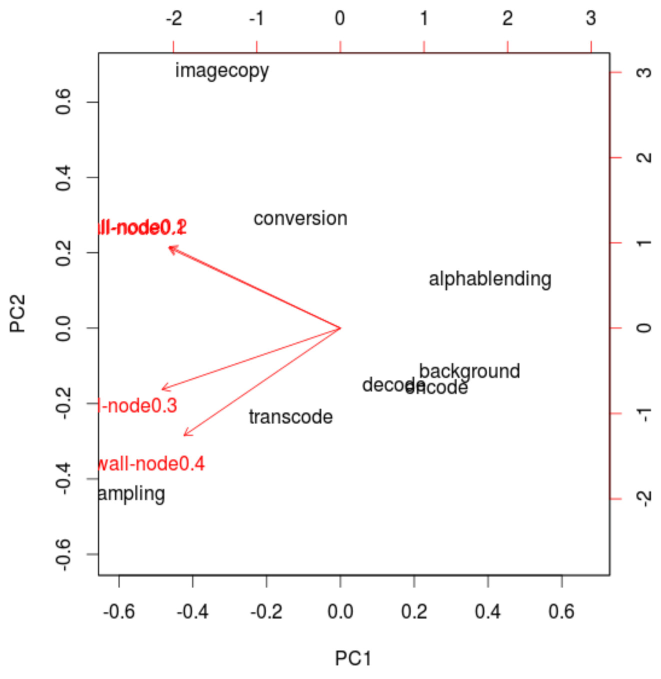


Figure 25 – PCA biplot of the FUSION benchmarks using the vwall node flavors.

Looking at the raw data samples and the resulting PCA transformation, one can conclude that the first PC is a representation of the absolute rate score: the lightweight services (e.g., resampling, imagecopy, etc.) have a higher absolute rate than the more heavyweight services (e.g., encode, background extraction). The alpha blending benchmark, though it being relatively lightweight, performs relatively badly on this machine and has the additional issue that its rate does barely increase with increasing cores. The second PC dimension roughly represents memory-intensiveness.

Applying this again to all machine nodes, results in the following plot, which looks very similar to the previous plot:

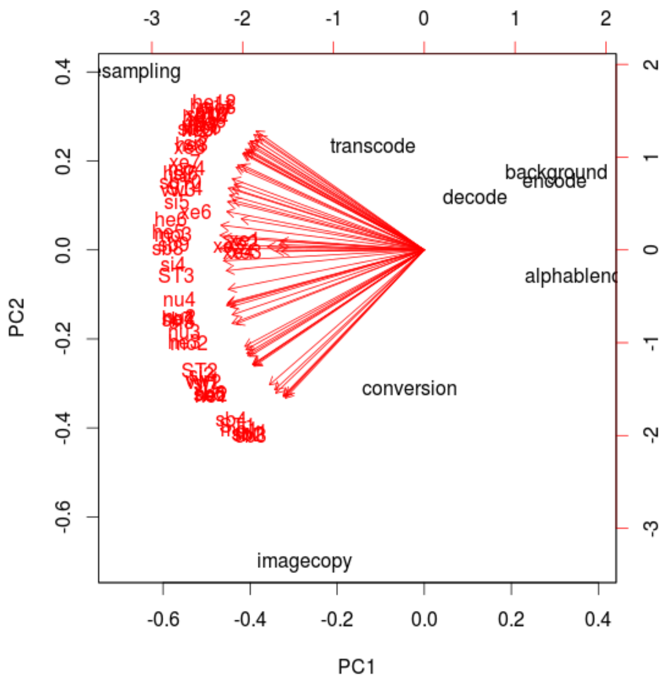


Figure 26 – PCA biplot of the FUSION benchmarks using all machine node flavors.

An interesting additional plot here is the corresponding factor loadings plot, as depicted in Figure 27. The parabola discussed earlier is very clearly visible in this plot. The xeonv2 machine however shows a clear deviation in PC2 behaviour, due to a different memory performance behaviour: on this machine, the rates increase much more linearly for memory-intensive applications than for the other machines.

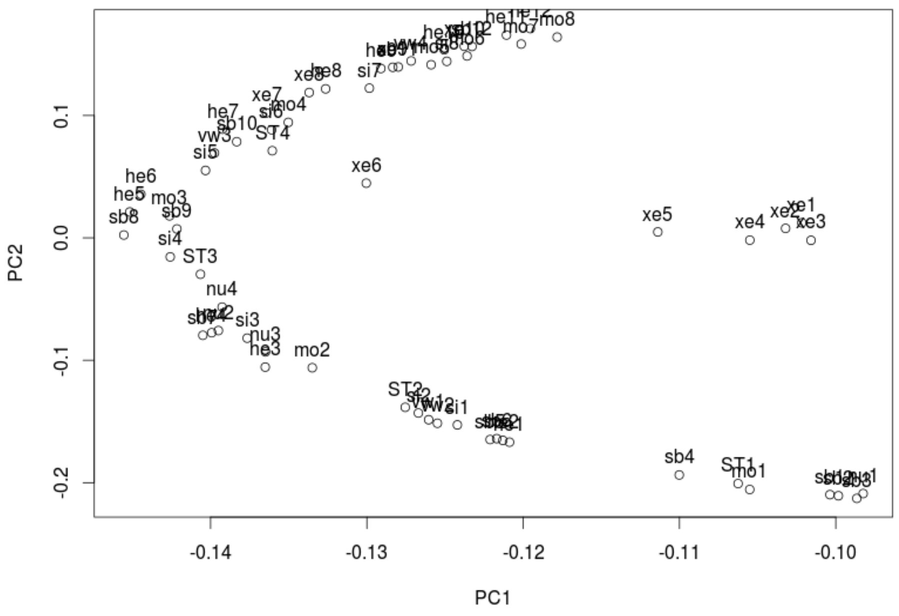


Figure 27 – Factor contributions of each machine node for the first two principle components for the FUSION benchmarks.

3.3.2.5 Future Work

In the previous sections, we focused on using PCA analysis for classifying application benchmarks and machine nodes or runtime environments. Many additional analyses can be performed w.r.t. the mpact of various heterogeneous virtualized runtime environments. From the application perspective, one could also use this technique to classify real-time applications.

We already started this work by using the performance predictability scores (see Section 3.3.4) instead of the throughput rates for CPU-intensive applications used in this section. An initial result graph is depicted in Figure 28 below, where we show the classification of the real-time fixed-frame rate variants of the FUSION media benchmarks. More work is needed to further clarify and refine these results.

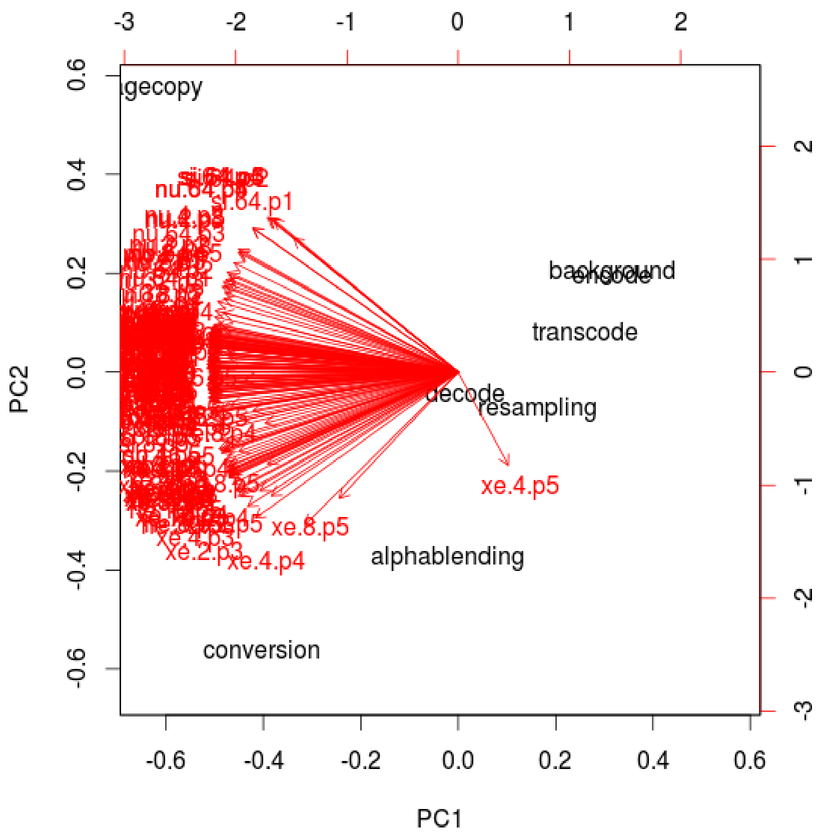


Figure 28 – PCA biplot of the real-time FUSION benchmarks using all machine node flavours and using the performance predictability scores for various predictability levels and system load levels.

3.3.3 Syscall Profiling

With this profiling method, the main idea is to measure the set of system calls an application performs, how dominant each set of system calls is, how long they take on a particular platform, etc, and use this information to help profiling the requirements and behaviour of a particular application, study how it behaves on a particular runtime environment, and correlate this information in order to understand what type of application it is, how it should be deployed, and estimate how it will behave on a similar runtime environment.

3.3.3.1 Motivation

In order to have a better understanding in the behaviour of a particular application in a particular runtime environment, we will use the system call traces of an application interacting with its encapsulating runtime environment to have a better understanding in what the application behaviour is w.r.t. its surrounding environment, but also how well an application responds to the runtime environment onto which it has been deployed.

On the one hand, this allows cloud providers to have a better understanding of the behaviour of a particular application without needing to have detailed descriptions or information on what that application needs or how sensitive it is w.r.t. its runtime environment. In other words, the application can still be treated as a black box, but through its interactions with the OS, we retrieve valuable knowledge of the type of application, its interactions and its sensitivities.

From an application provider perspective on the other hand, system call profiling gives additional detailed knowledge on how the surrounding runtime behaviour is behaving, how long particular system calls take, how stable the OS environment is for the application, etc. For example, on a best-effort highly oversubscribed environment, the syscall profile may look dramatically different than on a vanilla baseline environment, e.g. w.r.t. the relative timings profile of each system call (e.g., I/O calls take much longer and with intolerable variability). Hardware performance counters e.g. only give information on how efficient the CPU or memory system is for a particular application, but do not give any insight in why particular portions of the application suddenly start to slow down (e.g. due to some I/O calls taking much longer). In a multi-tenant environment, understanding the OS from an application perspective is very valuable.

3.3.3.2 Method

The overall method for syscall based profiling of an application in a particular runtime environment is as follows:

1) First, we monitor the OS system calls of the application and determine the amount of calls per system call function and time spent in the system call (OS call behaviour study).
2) Next, we select a baseline runtime environment and select a number of key benchmarks that characterize the 3 main resource types (i.e., compute, storage and IO).
3) We then run these benchmarks on the platform thereby registering the capabilities of the runtime environment, and study its OS call behaviour.
4) Fourth, we run the target application service on the baseline runtime environment and monitor the service's OS system call behaviour and obtainable service capacity. We also determine the proportional use of the 3 resource axes and compare these against the results from the key benchmarks. We do this for different/several service load conditions, inclusive the service's max capacity on the baseline environment. We subsequently register this information as a baseline service profile.
5) Then we run the target application service in a specific configuration setting or application load on the test platform and register the OS system call behaviour (distribution and time). Repeat for a second specific setting and register its profile.
6) Based on the results obtained in step 4, we now extrapolate the data from step 5 and estimate the maximum service load/capacity for the unknown platform.
7) We obtain the feedback results from the actual service running on the test platform and compare with the extrapolated results.

3.3.3.3 Benefits and Drawbacks

Some key benefits of this approach are the following:

- The approach to monitor OS system call activity (amount and time) has the benefit that it is a service agnostic method; applications can be profiled as a black box; only their interactions with the OS are monitored.

- No modification is necessary in the service itself since the monitoring occurs at the OS interface. Both the application as well as the cloud environment can trigger this type of profiling, though the cloud environment can typically do this much more efficiently (e.g., directly inside the OS kernel).
- In the cloud, there are numerous different working environments and due to various reasons e.g. from an economical point of view, benchmarking all of the environments becomes impractical. This is mainly in case you want to predict the behaviour of an application in a new unknown runtime environment without actually deploying the application.

This method however has also some limitations and drawbacks, meaning it is not sufficient nor always advised to rely on this profiling method alone:

- The amount of OS system calls are numerous (e.g. ~310 different functions for LINUX 3.x), so we need to cluster the various calls based on similarity (e.g. in line with the three main resource types).
- The overhead from call tracing causes additional load and resource usage onto the platform where the service is deployed, thereby reducing the max attainable service capacity obtainable without the additional call tracing load. For very syscall-intensive applications, this can even significantly slow down the overall application performance or even skew its behaviour.
- OS system call timing behaviour is influenced by various conditions like multi-tenancy, virtualization technique, etc., so it may not always be easy to gather a consistent or reliable profile on a particular environment.
- Running a service inside a VM and monitoring the VM OS system calls does not reveal any information about the service. The proposed method therefore is only suited for services running either on bare metal or containers.

3.3.3.4 Quali Tool

For measuring the syscall application profiles, we developed the Quali tool to capture, measure and postprocess the OS system call traces for a particular application.- The Quali tool uses *strace* as underlying mechanism for collecting the necessary information about each system call as well as the important timing information for each system call.

Our tool processes and summarizes all OS syscalls into a number of categories, acting as qualifiers, and calculates the relevant (average, stddev and tail) statistics for each of these qualifiers. Summaries of all OS calls, individually or grouped per qualifier are generated and presented afterwards.

For the evaluation results in the following sections, we used a fixed set of categories or qualifiers, grouping all system call functions into the following categories. Note that these could easily be refined further, with more relevant discriminating categories (e.g., memory, synchronization, etc.):

- COMM: all syscalls related to network communication (connect, listen, select, etc.)
- FILE: all syscalls related to file descriptor-based operations (read, write, etc.) Note that in this initial first implementation of Quali, we do not distinguish yet between the type of resource represented by the file descriptor (e.g., file, socket, etc.).
- FILECTRL: all syscalls related to control-related functions of file descriptors (e.g., seek, open, close, etc.)
- PROCESS: all syscalls related to Linux process management (e.g., clone, getpid, etc.)
- SLEEP: explicit timed sleep (nanosleep)
- UNDEF: all syscalls not part of any existing category
- WAIT: syscalls waiting for an external event to occur (e.g., epoll_wait, etc.)

Based on these statistics, a syscall profile can be constructed, along with the corresponding distribution, timing and tail latency graphs.

3.3.3.5 Methodology

3.3.3.5.1 Test Platform

For our experiments, we used a server with an ASUS Z9PE-D16 Series motherboard with 2 Xeon E5-2609v2@2.50Ghz, 8 cores, 2 NUMA nodes with 16GB + 8GB RAM (Hynix Semiconductor, HMT31GR7BFR4C-H) and QPI interconnect. Storage is Seagate ST1000DM003 (1TB with 6.0Gb/s). Evaluation Benchmarks

The benchmark applications that were used in the experiments are

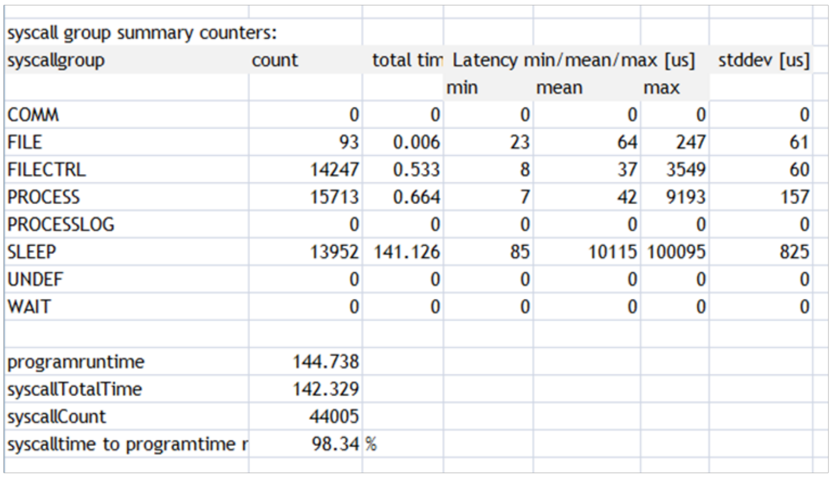
- Storage: FIO (Flexible Input/Output Tester Synthetic Benchmark, a versatile IO workload generator) [FIO13]
- Network: iperf3 (http://software.es.net/iperf/)
- Compute: in-house developed yuvconv and transcode, using the Vampire framework [Va09]

3.3.3.6 Evaluation Results

In the following paragraphs and subsections, we will present and discuss the first results and measurements done using each of the benchmarks on the reference platform. These results serve as initial insight into the feasibility, benefits and issues related to syscall based profiling.

3.3.3.6.1 FIO benchmark

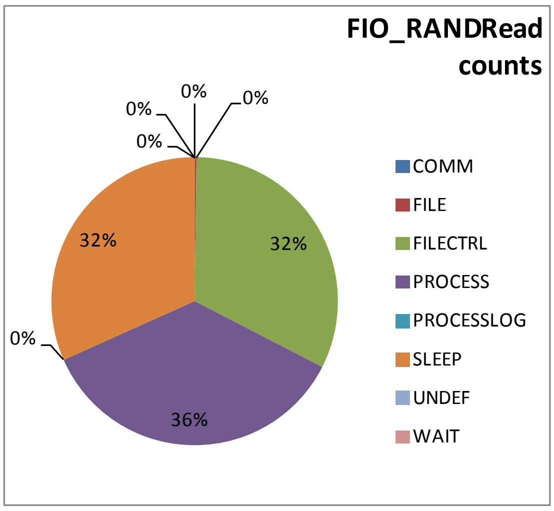
The FIO benchmark is a disk I/O intensive flexible benchmark application for evaluating the performance and runtime characteristics of a particular storage device and OS. For this benchmark, we assessed its behaviour for a 128 MB random read test. A summary of the results produced by our Quali-tool is depicted in Figure 29 for this benchmark on the reference platform. The count and time distribution of each qualifier is presented in Figure 30. *USER* represents the fraction of time not spent in system calls.



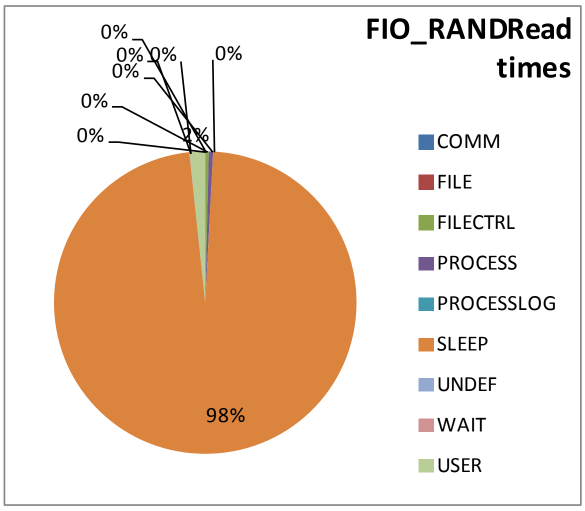
syscall group summary counters:						
syscallgroup	count	total tim	Latency min/mean/max [us]			stddev [us]
			min	mean	max	
COMM	0	0	0	0	0	0
FILE	93	0.006	23	64	247	61
FILECTRL	14247	0.533	8	37	3549	60
PROCESS	15713	0.664	7	42	9193	157
PROCESSLOG	0	0	0	0	0	0
SLEEP	13952	141.126	85	10115	100095	825
UNDEF	0	0	0	0	0	0
WAIT	0	0	0	0	0	0
programruntime	144.738					
syscallTotalTime	142.329					
syscallCount	44005					
syscalltime to programtime r	98.34	%				

Figure 29 – Summary of Quali syscall reporting for the FIO random read benchmark.

Firstly, it can be observed that almost all time (i.e., over 98%) is spent in the kernel for this benchmark. However, counter-intuitively, most of this syscall time is not spent in direct I/O related syscalls but in the nanosleep syscall that is part of the SLEEP category. Consequently, although this is an extremely disk I/O intensive benchmark, due to its asynchronous setup and specific implementation, this is not directly reflected in the syscall time distribution. Although in effect this application itself basically is running idle and most of the action is on disk, there is actually much more going on than visible from these graphs. As such, one important conclusion from this is that these syscall traces not necessary directly reflect the actual resource usage and dependencies; additional metrics are needed to better correlate the syscall patterns with the overall behaviour.



(a) Distribution of the number of syscalls per qualifier category



(b) Distribution of the total time spent in each syscall category or in USER space.

Figure 30 – Count and Time distributions of all syscalls for the FIO random read benchmark.

In Figure 31, we show the CCDF curves of the durations of each syscall category for this benchmark, representing the tail latencies for the various syscall groups. In this figure, one can observe that the application sleeps for about 10 ms per syscall.

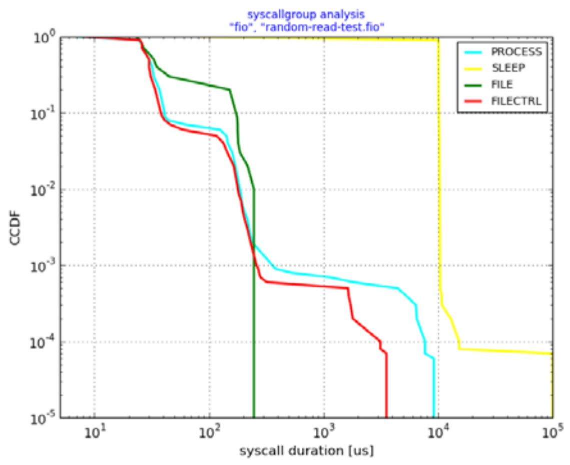
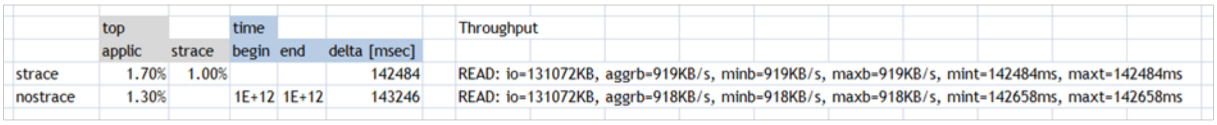


Figure 31 – CCDF curve of the duration of the OS system calls per category for the FIO benchmark.

Lastly, Figure 32 depicts the application throughput and CPU utilization and overhead when running the FIO benchmark with and without the tracing. The application is not impacted by the profiling overhead, as there are only a relative infrequent amount of system calls to be measured. For this benchmark, the FILECTRL overhead gives a rough idea of the absolute CPU load.

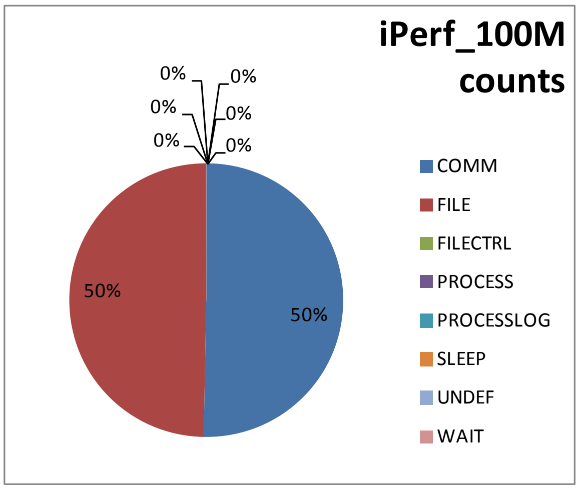


	top		time			Throughput
	applic	strace	begin	end	delta [msec]	
strace	1.70%	1.00%			142484	READ: io=131072KB, aggrb=919KB/s, minb=919KB/s, maxb=919KB/s, mint=142484ms, maxt=142484ms
nostrace	1.30%		1E+12	1E+12	143246	READ: io=131072KB, aggrb=918KB/s, minb=918KB/s, maxb=918KB/s, mint=142658ms, maxt=142658ms

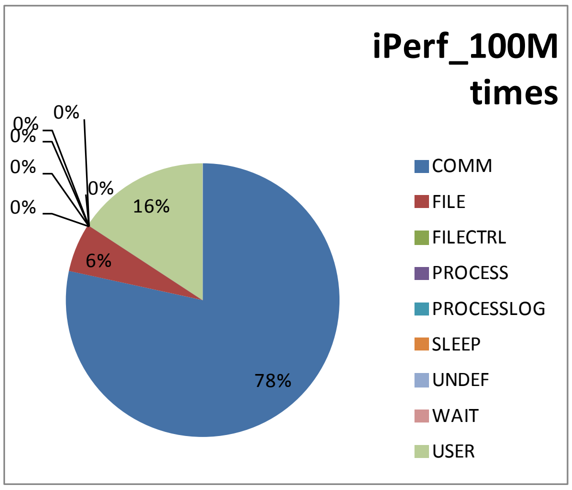
Figure 32 – Overhead of the syscall tracing tool w.r.t. the application being profiled.

3.3.3.6.2 iPerf3 benchmark

This benchmark measures the networking performance in between two hosts. In this setup we configured the benchmark as a client on different machines interconnected by either a 100Mb or 1Gb switch. The overall count and timing distributions of the various syscall categories are depicted in Figure 33 for a 100 Mb switch and in Figure 34 for a 1Gb switch. W.r.t. the relative number of syscalls, there is an equal share between the FILE and COMM categories, whereas COMM is mainly the *select* syscall, and FILE is mainly the *write* syscall. In the 100Mb case, most of the application time is actually spent into this COMM category, meaning the application is mainly waiting for sending additional packets to the remote server. Only 16% is spent in user space. The *write* syscalls in the FILE category only consume about 16% of the overall application time.



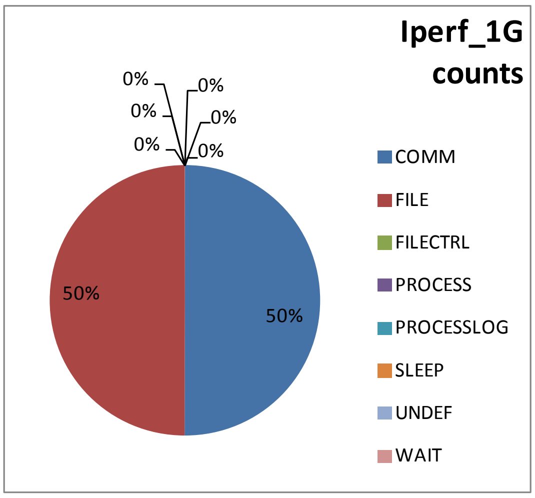
(a) Distribution of the number of syscalls per qualifier category



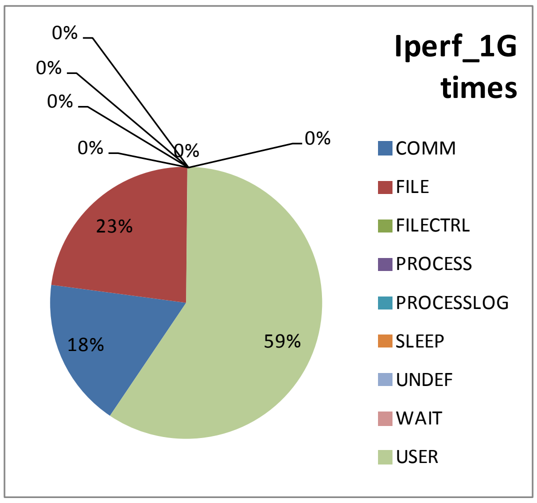
(b) Distribution of the total time spent in each syscall category or in USER space.

Figure 33 – Count and Time distributions of all syscalls for the iPerf3 benchmark for a 100Mb switch.

For the 1G case, although one can still observer the equal share of COMM and FILE syscall events, the relative timings are very different. First, there is an unexpected higher amount of user space time, consuming about 60% of the total application execution time. This will be explained when discussing the syscall tracing overhead in the following paragraph. However, also the relative timing between the FILE and COMM categories is very different here. This is because the packet rate is much higher in this case, so less time is spent in waiting for the previous packets to be processed.



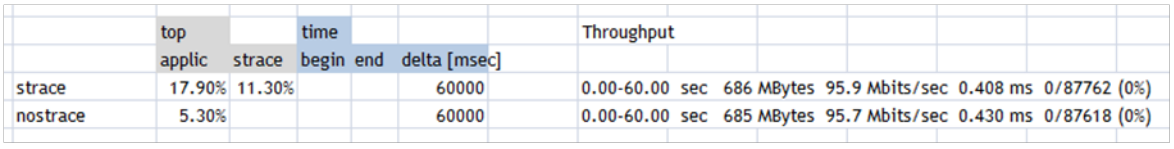
(a) Distribution of the number of syscalls per qualifier category



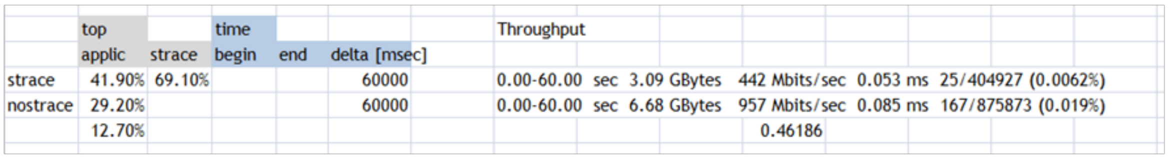
(b) Distribution of the total time spent in each syscall category or in USER space.

Figure 34 – Count and Time distributions of all syscalls for the iPerf3 benchmark for a 1Gb switch.

In Figure 35, the actual application throughput results as well as the overhead of *strace* is presented, both in the 100Mb (upper graph) and in the 1Gb case (lower graph). In the 100Mb case, the iperf3 benchmark effectively achieves about 96Mb/s throughput, and the overhead of *strace* is about 11% additional CPU time. The application itself however is not impacted. In the 1Gb case however, the overhead of the default *stracing* has a significant impact on the application performance. Although the application reaches the 1Gb/s when tracing is disabled, in the tracing case, only 442 Mb/s actual throughput is achieved. The core reason is the overhead of *strace*, which consumes about 69% of CPU time, leaving not enough CPU time to reach the target 1Gb/s packet rate. In this case, about 855k syscalls are being recorded for the duration of the test application (i.e., 60 seconds), resulting in a syscall rate of almost 15k per second. Consequently, to be able to effectively use a syscall tracing tool such as Quali, statistical sampling should be used to reduce the overall overhead. Especially in this case, due to the highly repetitive nature of the syscall pattern, only a fraction needs to be captured in order to understand its average or outlier behaviour. Different techniques could be used for this [VE09].



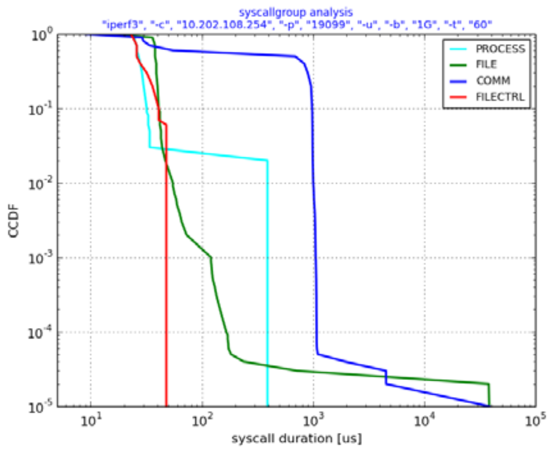
	top		time			Throughput
	applic	strace	begin	end	delta [msec]	
strace	17.90%	11.30%			60000	0.00-60.00 sec 686 MBytes 95.9 Mbits/sec 0.408 ms 0/87762 (0%)
nostrace	5.30%				60000	0.00-60.00 sec 685 MBytes 95.7 Mbits/sec 0.430 ms 0/87618 (0%)



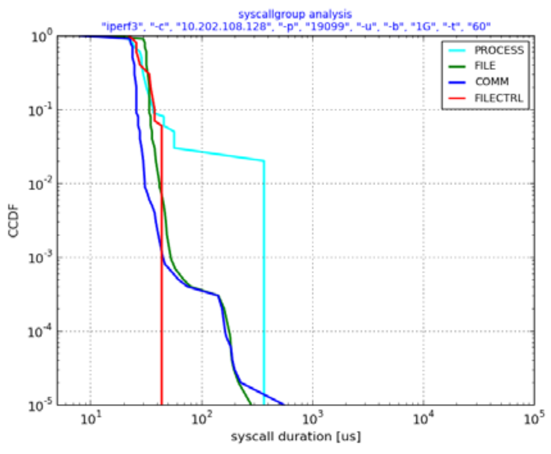
	top		time			Throughput
	applic	strace	begin	end	delta [msec]	
strace	41.90%	69.10%			60000	0.00-60.00 sec 3.09 GBytes 442 Mbits/sec 0.053 ms 25/404927 (0.0062%)
nostrace	29.20%				60000	0.00-60.00 sec 6.68 GBytes 957 Mbits/sec 0.085 ms 167/875873 (0.019%)
	12.70%					0.46186

Figure 35 – Overhead of the syscall tracing tool w.r.t. the application being profiled, for a 100Mb switch (upper graph) and a 1Gb switch (lower graph).

The tail behaviour of both test cases is shown in Figure 36. Whereas in the 100Mb case, the duration of the COMM category dominates the other syscall timings, for the 1Gb case, the FILE and COMM durations go hand in hand.



(a) 100Mb switch

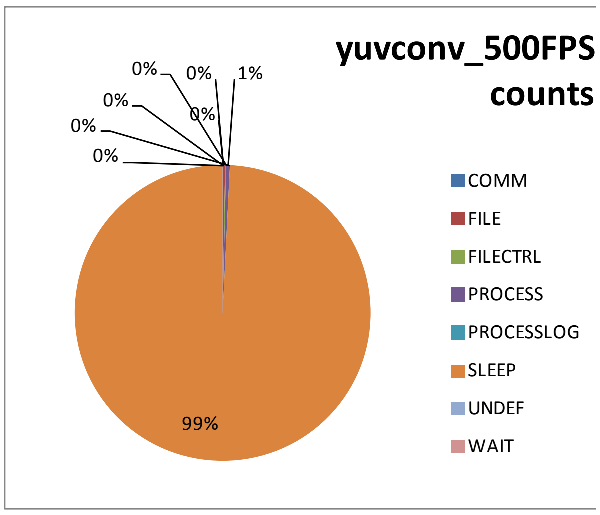


(b) 1Gb switch

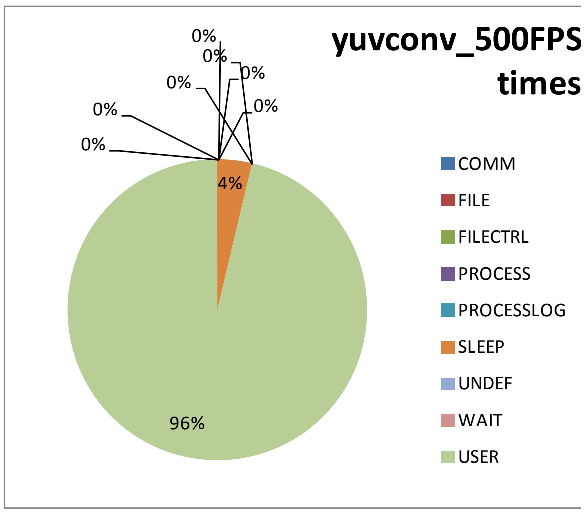
Figure 36 – CCDF curves of the duration of the OS system calls per category for the iPerf3 benchmark, for a 100 Mb and 1Gb switch, respectively.

3.3.3.6.3 YUVConv benchmark

This benchmark is a custom-created memory and compute intensive benchmark in our Vampire framework. It includes almost zero syscall activity, except at the beginning and end of the application. The relative count and time duration distributions as depicted in Figure 37. W.r.t. the number of syscalls, it is clear that the SLEEP syscall is the only relevant syscall in this benchmark. However, at 500 FPS, the time spent sleeping is only a tiny fraction of the overall execution time (i.e., 4%). At lower frame rates, the time spent sleeping would increase accordingly. As such, for this benchmark, one could conclude that the maximum CPU capacity is almost reached. By observing the SLEEP syscall behaviour instead of or combined with the actual CPU utilization, one can have a deeper knowledge in the true bottleneck of the system. For example, 100% CPU utilization not necessarily corresponds to a saturated system in case of a multi-threaded application. By observing the SLEEP trace (tail) latency behaviour, one can possibly obtain a better understanding in runtime behaviour and bottlenecks.



(a) Distribution of the number of syscalls per qualifier category



(b) Distribution of the total time spent in each syscall category or in USER space.

Figure 37 – Count and Time distributions of all syscalls for the YUVConv benchmark.

The tail latency behaviour of the syscalls (especially the SLEEP category) is depicted in Figure 38. One clear observation is that the tail curve of the SLEEP syscall group is not a vertical line, but a curved line, meaning there is variation in the amount of time it takes to process each frame.

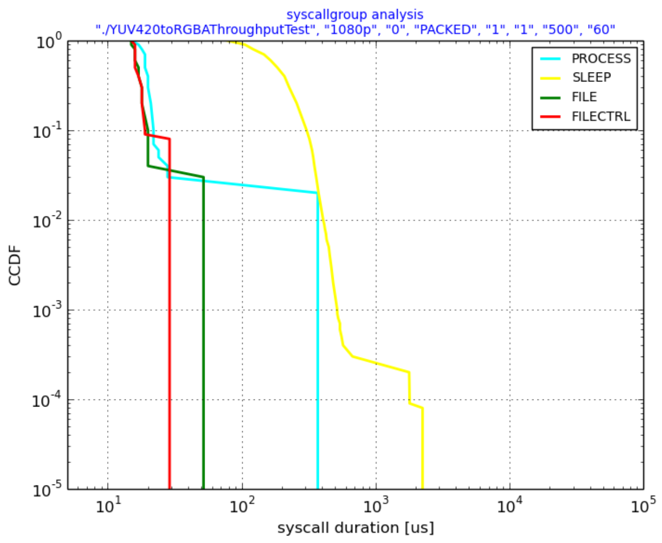


Figure 38 – CCDF curve of the duration of the OS system calls per category for YUVConv benchmark.

3.3.3.6.4 Videotranscode benchmark

This benchmark is an in-memory video transcode benchmark that transcodes some input video file, downscale the video to a lower resolution and encodes the new video stream. As this benchmark is also configured as a compute-intensive test, not including extensive network and/or file I/O. As such, the syscall behaviour looks very similar to the previous benchmark (although the functional behaviour is very different), with similar conclusions. For this, we did not include the detailed graphs for this benchmark.

3.3.3.7 Conclusions and Future Work

In this section, we presented a syscall-based profiling technique to measure the behaviour of an application in its interactions with the OS, and use this to help assessing the impact of an environment and changes in the environment onto the system behaviour of the application. For this we developed a tool called Quali, and applied this onto a range of application types.

In conclusion, the approach resulted a rather coarse-grained, may have a significant overhead if active 100%, and does not always produce intuitive results. Consequently, this profiling and/or monitoring technique alone definitely not is not sufficient to estimate session slot capacity, though it can be used to fine-tune session slot capacity, as it can be used for measuring OS runtime sensitivity, and used in combination with other profiling and/or monitoring information and tools. Also, for syscall-intensive applications, a sampling technique should be used instead.

Future work concerning this topic may involve the following topics:

- Refine categories, possibly dynamically;
- Reduce syscall tracing overhead via dynamic sampling;
- Include a sensitivity analysis based on the timings;

- Extrapolation of expected runtime behaviour of an application on a new platform based on results from specific benchmarks, combined with sensitivity analysis.

3.3.4 Performance & Predictability Curves

This section discusses a visualization method for representing the performance and *predictability* behaviour of various *sensitive* applications in across different runtime environments.

3.3.4.1 Motivation

Cloud infrastructures typically focus on providing a particular average performance or throughput or energy efficiency for particular types of applications. Although this is sufficient for a wide range of (cloud) applications, for more *sensitive* applications such as real-time applications, the overall QoS provided by these infrastructures is equally important.

In this section, we define a **sensitive application** as an application for which a particular average performance or throughput (e.g., CPU, memory, storage, networking, etc.) is not sufficient, but for which the amount of variability in delivering that performance is equally (if not more) important. For example, for a low-latency real-time transcoding service, it is not only crucial that a particular average performance level is reached, but also the consistency in terms of scheduling and overall responsiveness is equally important. If that service is not scheduled for a long time, then deadline misses will occur, causing dropped frames and/or jitter.

Consequently, the *predictability* of a particular runtime environment for this class of sensitive applications is equally important. We define **predictability of a runtime environment** as the level of consistency in runtime behaviour that such environment can provide for a particular application. The required level of predictability strongly depends on the application. For a typical IT service, average throughput and some low degree of consistency is typically sufficient. For more stringent real-time interactive low-latency services, the predictability requirements towards the runtime environment drastically increase.

To be able to visualize, reason and compare the performance and predictability characteristics of a particular environment (especially in a heterogeneous environment), we came up with the concept of performance predictability curves and regions, which is discussed in detail in the following sections.

3.3.4.2 Concept

The effective performance and predictability of a particular runtime environment are typically not independent from each other, but are rather negatively correlated: the more performance is squeezed from the environment, the less predictable the environment becomes. For example, deploying only a single single-threaded application on a particular (physical) runtime environment typically will have excellent predictability (when configured appropriately), but delivers poor performance, as the environment typically will be underutilized. On the other hand, if more and more services are scheduled onto that same environment, the effective performance of the environment increases, but at the expense of reduced predictability due to the multi-tenancy and imperfect resource sharing: scheduling issues, caching issues, virtualization issues, etc.

A conceptual drawing of this trade-off in performance and predictability, is depicted in Figure 39. The blue curve in this graph represents a typical trade-off in performance and predictability for a typical cloud node: the more performance is squeezed out of a cloud node, the lower the corresponding predictability due to increased runtime jitter and resource stalls.

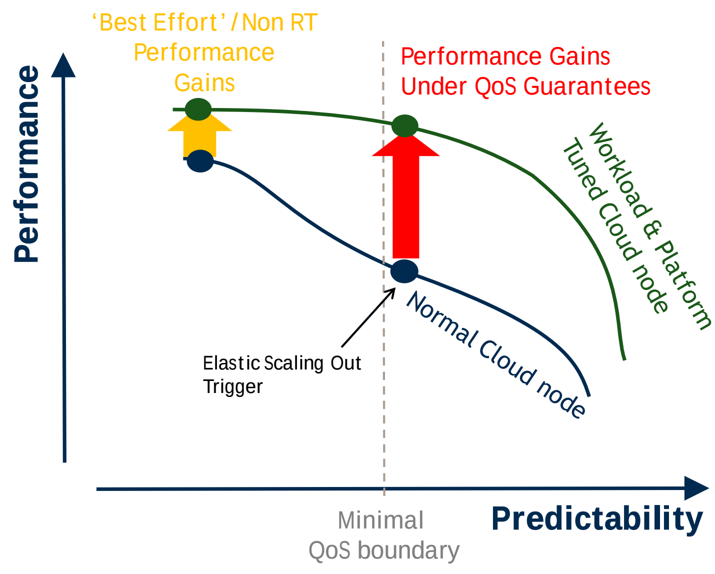

'Best Effort' / Non RT Performance Gains
Performance Gains Under QoS Guarantees
Performance
Workload & Platform Tuned Cloud node
Normal Cloud node
Elastic Scaling Out Trigger
Minimal QoS boundary
Predictability

Figure 39 – Conceptual graph of the performance and predictability relationship.

For non-real-time best-effort cloud applications, this is typically not much of a problem (until some point obviously), and maximizing aggregate performance or throughput is typically the main objective to reduce TCO or maximize profit. This is represented in the left dot on the blue curve.

Sensitive or real-time applications on the other hand have some minimum predictability boundary below which the QoS provided by the application would be insufficient (e.g., resulting in too many deadline misses or too long delays). When such application would be scheduled on that same basic cloud node, one could only increase the aggregate performance of that node until the minimal QoS boundary is reached. This is represented in the right dot on the blue curve. Instead of trying to increase the application load even further on such environment, a new instance would have to be spawned on another node. As such, the maximum achievable performance would be much lower than for a best-effort application.

Imagine however an optimized heterogeneous cloud platform that takes into account the capabilities and limitations of a runtime environment and the requirements of the applications, then a higher performance (efficiency) can be reached for both the best-effort as well as the real-time/sensitive applications. This behaviour is represented by the green curve on the graph. Especially for the latter class of applications, when done correctly, such workload and platform tuned cloud node could drastically improve overall performance (efficiency) while still guaranteeing the minimal application QoS.

Note that performance can be represented in different ways, such as aggregate throughput or density, energy efficiency, or cost efficiency. A conceptual example of this is depicted in Figure 40. For particular types of applications, using different hardware nodes can be more energy and/or cost efficient, and may result in better predictability for particular applications. Hardware accelerators or micro-server architectures can provide significant boosts in performance/energy/cost efficiency, though not for all applications, and in many cases, there will also be a minimal economical boundary below which using these hardware infrastructures is not viable (though perhaps more predictable).

In the following sections, we will further explore these performance-predictability curves (PPCs) for some concrete real-time test applications and platforms.

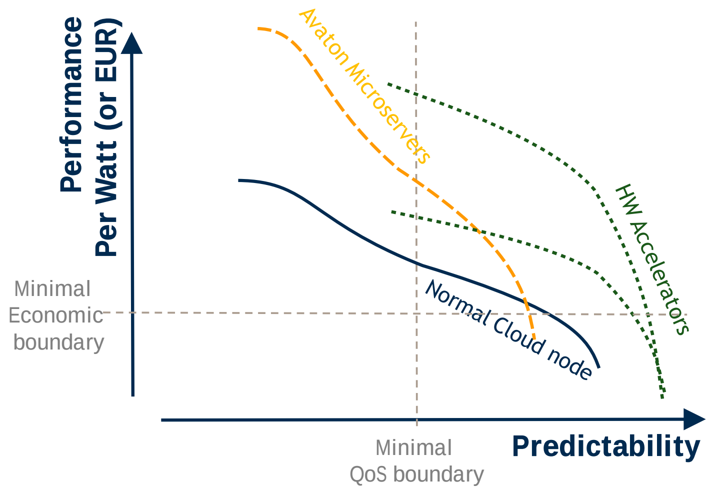


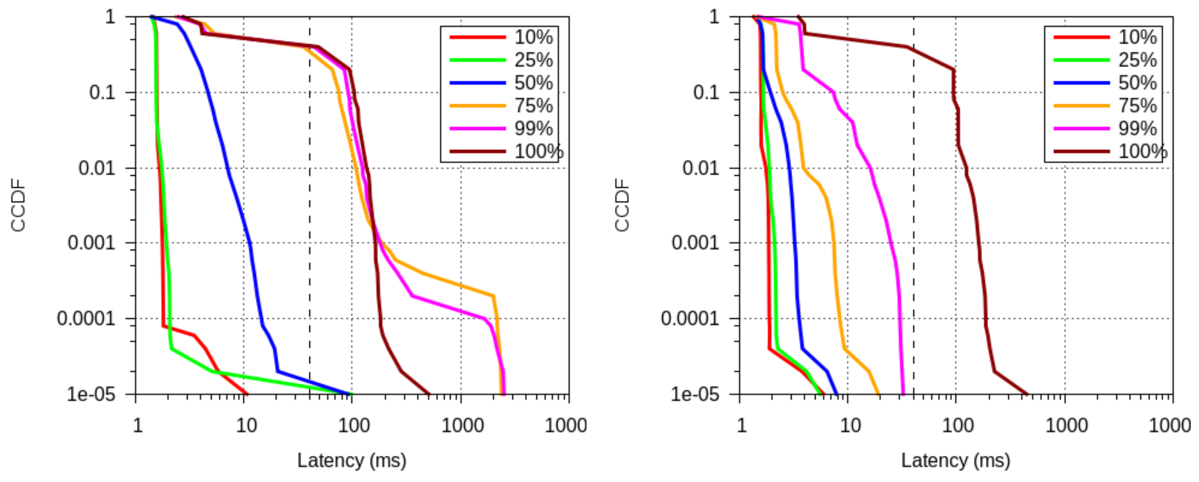
Figure 40 – Conceptual graph of the trade-off in energy/cost efficiency with predictability for different hardware environment types.

3.3.4.3 CCDF Curves

A fundamental question is how to capture and represent 'predictability' for a particular (QoS-sensitive) application that is deployed *in some manner* (e.g., with/without virtualization, with a particular OS and OS settings, using a particular SW configuration, etc.) *on some hardware* (e.g., Xeon vs Atom, with or without particular HW accelerators, etc.).

Ideally, we would like to place the performance-predictability curves of (different) applications deployed in different manners on different hardware on the same graph for analysis and comparison. Obviously, the end results should still be a valid and meaningful graph, so that if two points on two different curves have the same predictability score, that their actual inherent predictability should be the same as well (e.g., QoS/QoE, failure rate, etc.). We thus need a metric for predictability that is valid across these changing configuration settings.

In case of real-time interactive low-latency media applications, the tail latency behaviour is very important. As tail latency behaviour is very clearly represented in CCDF curves, we will use these curves as a basis for representing predictability. Two example CCDF plots are depicted in Figure 41.



(a) Out-of-the-box deployment on host. (b) Application-optimized deployment on host.

Figure 41 – CCDF latency curves of a real-time image processing application on a test platform, in case of (a) an out-of-the-box deployment, and (b) an optimized deployment.

In these graphs, each curve represents the tail latency behaviour under a particular relative system load (i.e., 10%, 25%, ..., 100%). The application latency is defined as the amount of time it takes to process an entire video frame. As the target frame rate for this test application is set to 25 FPS, the maximum latency before deadline misses start occurring, is 40 ms, as represented by the dashed vertical line. These CCDF plots show the probability that the latency is at least *X* ms. The left graph shows the tail latency behaviour under different (normalized) loads in case of an application-agnostic out-of-the-box deployment of the virtualized application on the host, whereas the right graph shows the behaviour in case application-aware optimizations are done (e.g., NUMA placement) when deploying the application onto the same runtime environment. System load is increased by starting more instances of the same application, representing an increasing amount of used session slots.

Two clear observations from these graphs are the following. First, the tail latency behaviour clearly deteriorates as the relative load is increased on the system. In other words, there is a higher probability that a particular maximum application latency will occur; the scenario with the higher load has a wider tail. In the non-optimized case, the tail latency is already unacceptable for relative loads above 50%. In the former case, the tail latency remains below the 40 ms barrier, whereas in the non-optimized latter case, the tail latency is already unacceptable for relative loads above 50%.

We now want to summarize such CCDF curve into a predictability score, capturing the position, orientation and/or shape of these curves. The position of these curves will be summarized in performance-predictability curves, whereas the orientation/shape of these CCDF curves will be summarized in performance-predictability regions.

3.3.4.4 Performance-Predictability Curves

Measuring an application's predictability score depends on the target QoS a particular application service provider has set for a particular service deployment. Even for the same application code, a service provider can choose to deploy the same application with different QoS settings: a cheap basic game server with occasional frame drops, and a premium version with virtually zero frame drops. Consequently, the performance-predictability curves will be dependent on the specific QoS level chosen by the service provider.

In our examples, a target QoS level of 90% is defined as the tail latency behaviour for the 90% quantile of all latencies. In other words, we are interested in the 10% worst behaviour: on average once every 10 frames (i.e. 0.4 seconds), the latencies will be *X*. The predictability at that point then is defined as the ratio between the maximum acceptable latency and the tail latency at that 90% quantile. In other words:

$$P_{QoSlevel} = \frac{Latency_{max}}{TailLatency_{QoSlevel}}$$

In our example, the maximum acceptable latency $Latency_{max}$ is set to 40 ms (though in real life this would even be much smaller, as the roundtrip latency may be much lower than 40 ms, even when the actual frame rate is only 25 FPS). If the tail latency is 40 ms at the 90% QoS level, then the predictability score is 1, meaning it just meets the target expectations. If the tail latency is 80 ms at that point, then the predictability score is 0.5, meaning it does not meet the target requirements by a factor 2. If on the other hand the tail latency is only 20 ms at that target QoS level, then the predictability score is 2, meaning the environment is twice as predictable/stable as minimally required.

Table 1 summarizes a number of QoS levels and the corresponding possible mean time between failure (MTBF) for real-time media applications running at various frame rates. For basic quality media applications running at low frame rates, lower QoS levels can be sufficient, whereas for premium quality high frame rate real-time media services (e.g., a high-end VR service), very high QoS levels may be required for which the predictability score needs to be at least 1.

QoS Level	25 FPS	50 FPS	100 FPS
90%	0.4	0.2	0.1
99%	4	2	1
99.9%	40	20	10
99.99%	400	200	100
99.999%	4000	2000	1000

Table 1 – Translation of various QoS levels into MTBF for various frame rates: the average number of seconds in between possible failures (e.g., frame miss or drop in case of high tail latency).

In Figure 42, an example performance-predictability curve is shown, for a target QoS level of 99.999%, using the CCDF curves (and others) for calculating the predictability score at different application load levels for different virtualization mechanisms and optimization strategies. In this graph, *bare* represents a bare metal deployment, *kvmbare* represents a single KVM-based fat VM in which all application instances are deployed, whereas *kvm* represents individual KVM-based VM instances per active application instance. *Numa* represents a set of NUMA-aware memory optimizations when deploying the application on the host or VM.

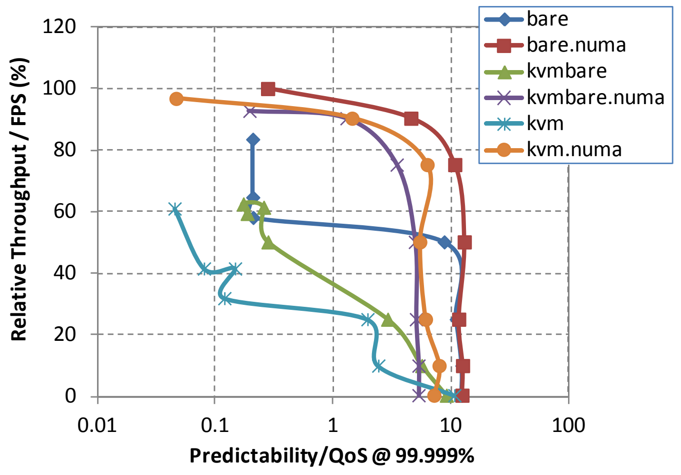


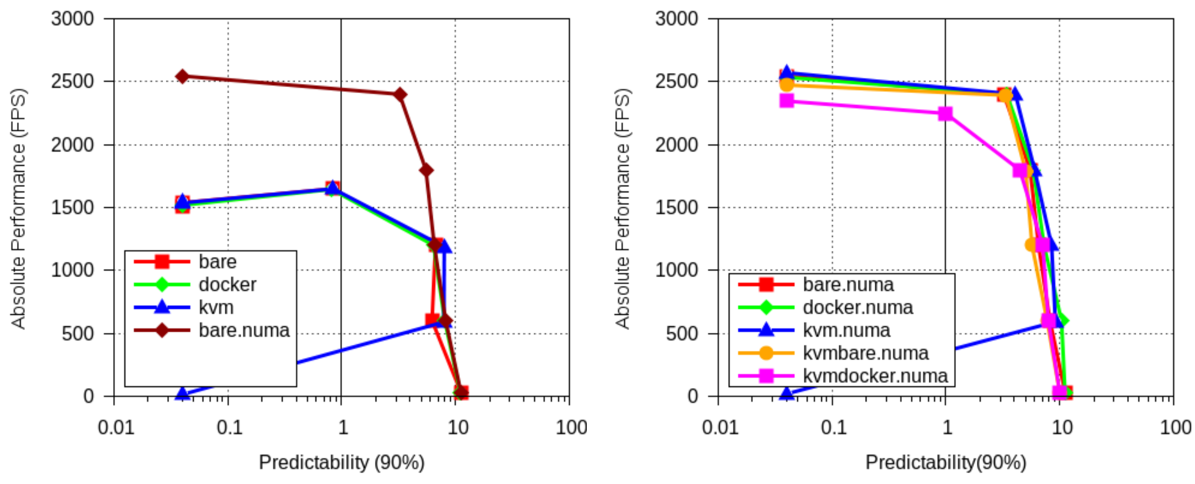
Figure 42 – Example Performance-Predictability Curve for a YUV conversion benchmark with a target latency QoS level of 99.999%.

This graph shows the trade-off between predictability and aggregate throughput, which is here represented as a percentage of the maximum achievable throughput on that environment. In order to meet the target QoS level of 99.999%, the load can only be increased up until the point where the curve passes the predictability threshold of 1. As long as the predictability score remains above 1, the target QoS level of 99.999% (which corresponds to a missed frame once every 1.1 hours at 25 FPS) is achieved.

As can be observed, the virtualization strategy as well as deployment optimization strategy has a significant impact on the maximum achievable aggregate throughput. Even when disregarding predictability, the maximum best-effort throughput varies significantly depending on the deployment strategy of the application. Deploying applications in individual virtual machines without any optimizations (i.e., *kvm*) has the worst performance and predictability curve: only about 25% of the optimal throughput can be achieved when taking into account predictability, and about 60% when disregarding predictability (e.g., in case this would be an offline parallel transcoding service). Applying application-specific optimizations however can significantly boost the overall performance and predictability relationship (i.e., *kvm.numa*) to achieve almost-optimal throughput (both with and without predictability). The predictability margin is not as high as for example in case of bare.numa (i.e., a predictability score of about 5.5 compared to about 11), but still more than sufficient for the target QoS level.

In the following paragraphs, we show a number of additional performance-predictability curves. These experiments were run on an Opteron 24-core, running the same YUV conversion benchmark at 25 FPS, with skipping missed frames enabled, running on a Linux 3.11 kernel. We show both the non-optimized and optimized cases, and include the results of deploying the application instances in Docker containers. Unless otherwise stated, the performance results a shown in terms of absolute FPS.

In Figure 43, the results are shown for a target QoS level of 90%. For this low target QoS level, the main difference in performance and predictability is between the non-optimized and optimized cases. Interestingly however, *kvm* seems to have a predictability issue when running a single instance of the application. This turns out to be some start-up overhead issue that was very specific to the 3.11 Linux host OS kernel version and did not appear in other kernel version. Evaluator services could detect these issues, while internally generating these curves to estimate session slot availability.

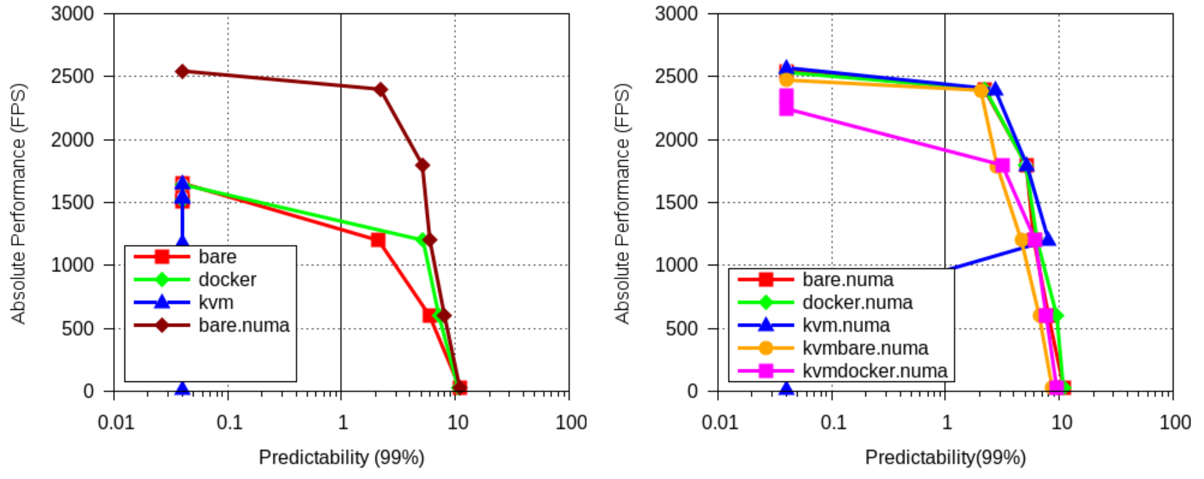


(a) Out-of-the-box deployment on host.

(b) Application-optimized deployment on host.

Figure 43 – Performance-Predictability Curve for a YUV conversion benchmark with a target QoS level of 90%, in case of (a) an out-of-the-box deployment, and (b) an optimized deployment.

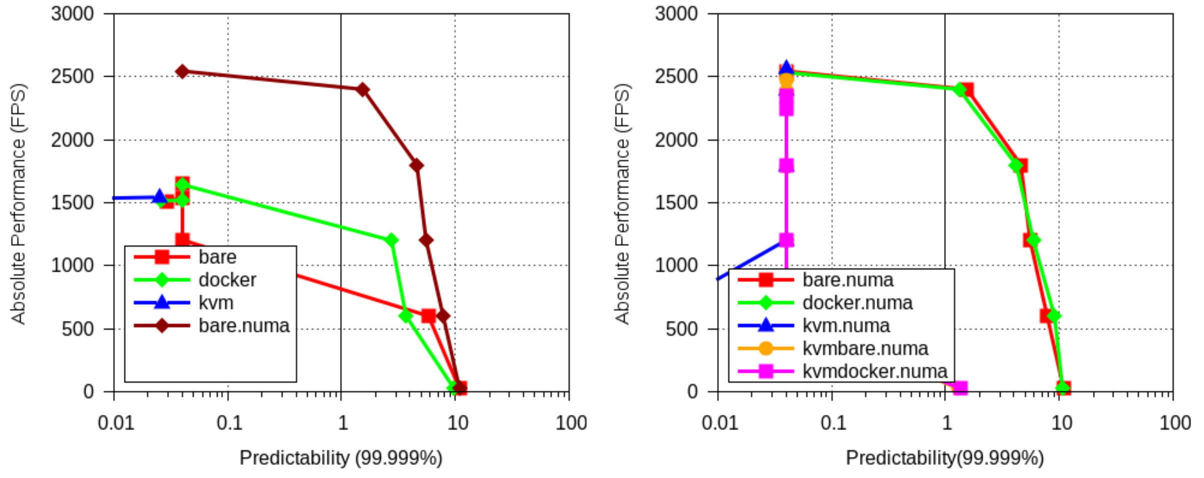
In Figure 44, the same results are depicted for a target QoS level of 99%. With this threshold, the issue with *kvm* seems to be even worse, especially in the non-optimized case, but also in the optimized case. The containerized versions on the other hand are on par with the bare metal variant.



(a) Out-of-the-box deployment on host. (b) Application-optimized deployment on host.

Figure 44 – Performance-Predictability Curve for a YUV conversion benchmark with a target QoS level of 99%, in case of (a) an out-of-the-box deployment, and (b) an optimized deployment.

Finally, in Figure 45, the results are shown for a target QoS level of 99.999%. Note that a dropped frame is represented as a frame with a 1000ms latency, hence the concentration around predictability score 0.04. The *kvm* results here are way off scale; the Docker container results on the other hand are as-good-as or even better than bare metal. Running (containerized) application instances in one fat VM (as typically would be the case in a cloud overlay mode) results in poor predictability at this QoS level, even in the optimized case, with too many dropped frames.



(a) Out-of-the-box deployment on host. (b) Application-optimized deployment on host.

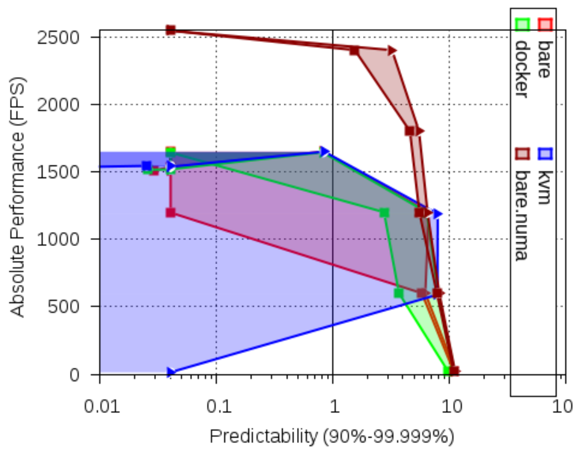
Figure 45 – Performance-Predictability Curve for a YUV conversion benchmark with a target QoS level of 99.999%, in case of (a) an out-of-the-box deployment, and (b) an optimized deployment.

Although we have many more results, running different benchmarks under different conditions and with different settings, we will skip these to save space. Basically, they all show similar results with similar conclusions.

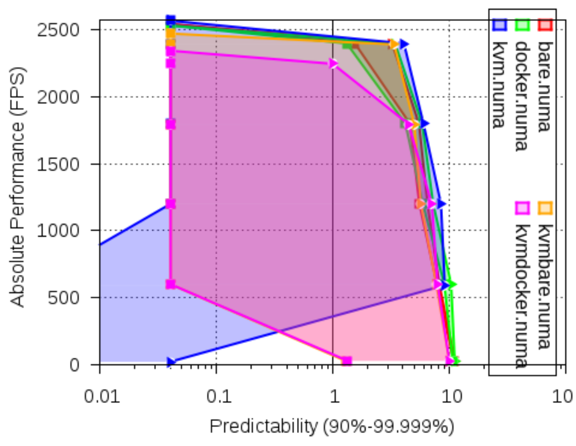
3.3.4.5 Performance-Predictability Regions

In the previous section, we introduced performance predictability curves as an intuitive visual representation of the performance and predictability trade-off of a particular runtime environment for a particular sensitive application with particular QoS requirements. As stated before, apart from using these graphs to visualize and reason about the impact of various runtime environments, heterogeneous cloud optimization, etc., this methodology can also be used by evaluator services for evaluating particular runtime environments and determine the maximum number of useful session slots. A cloud infrastructure provider could also use this to assess the efficiency of its environment for particular applications, provided it can retrieve or estimate the predictability scores from these environments.

However, to summarize the sensitivity of particular applications deployed in particular runtime environments, we also want to capture the orientation and/or shape of the corresponding CCDF curves and present them in a similar visual manner. This is achieved by not only showing the results of a single target QoS level in these graphs as curves, but to show the results for a target QoS range as filled regions. This is depicted in Figure 46, where we show such graphs using the same settings as used in the previous section, for a target QoS region of 90-99.999%. In these graphs, the width gives a clear idea of the sensitivity of an environment for different target QoS levels. For example, *bare.numa* and *docker.numa* are very narrow and vertical regions, representing very stable and predictable performance, whereas in the other *kvm**-cases, the predictability regions are huge, indicating various issues. Notice also the larger predictability regions (as well as reduced absolute performance) for *bare* and *docker* compared to the NUMA-optimized variants.



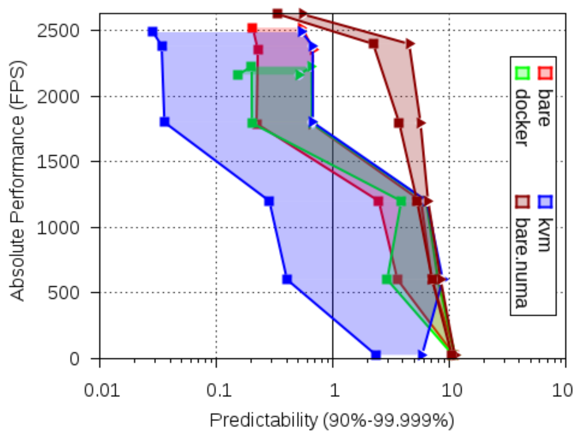
(a) Out-of-the-box deployment on host.

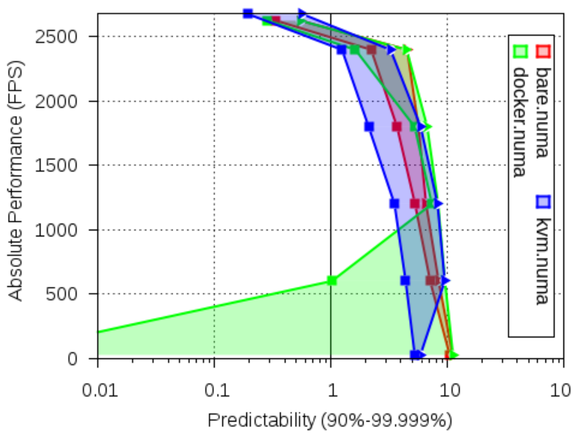


(b) Application-optimized deployment on host.

Figure 46 – Performance-Predictability Region for a YUV conversion benchmark with a target QoS range of 90-99.999%, in case of (a) an out-of-the-box and (b) an optimized deployment.

In Figure 47, the results are shown in case of a 3.8 kernel instead of a 3.11 kernel. The difference in predictability regions is quite substantial. The *kvm* virtualization strategy for example performs much better here, also w.r.t. the absolute performance in case of the non-optimized deployment scenario; the 3.8 kernel on the other hand is apparently less beneficial for Docker, as in the optimized case, predictability seems to be an issue for low system load.

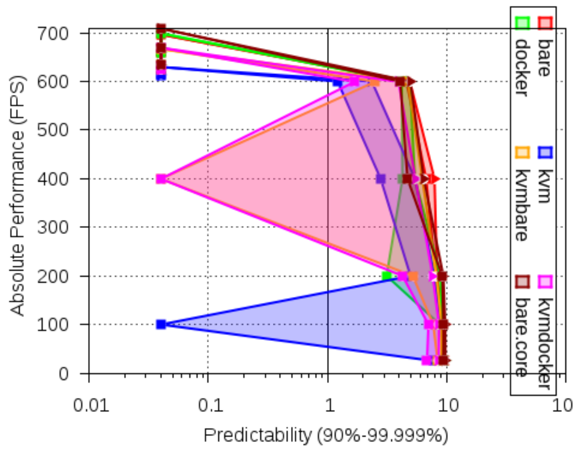


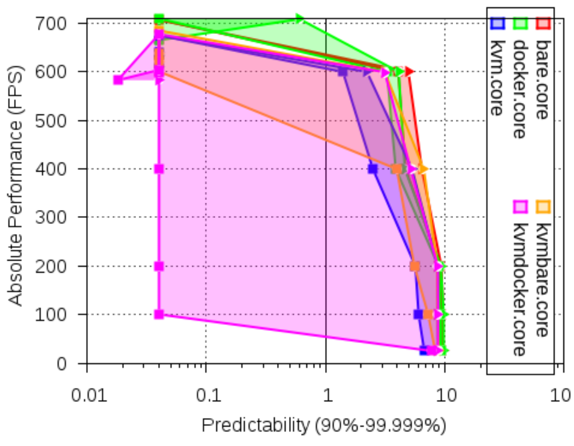


(a) Out-of-the-box deployment on host. (b) Application-optimized deployment on host.

Figure 47 – Performance-Predictability Region for a YUV conversion benchmark with a target QoS range of 90-99.999%, in case of (a) an out-of-the-box and (b) an optimized deployment, in case of a 3.8 kernel instead of a 3.11 kernel.

All results so far were done on a Opteron 24-core server. Figure 48 depicts the results on an Avoton Intel Atom micro-server with 8 lightweight CPU cores. Apart from the highly reduced absolute performance, there is also the obvious difference in predictability behaviour depending on whether application-specific optimizations are done or not. All of this impacts the effective amount of session slots that can be announced with particular QoS.



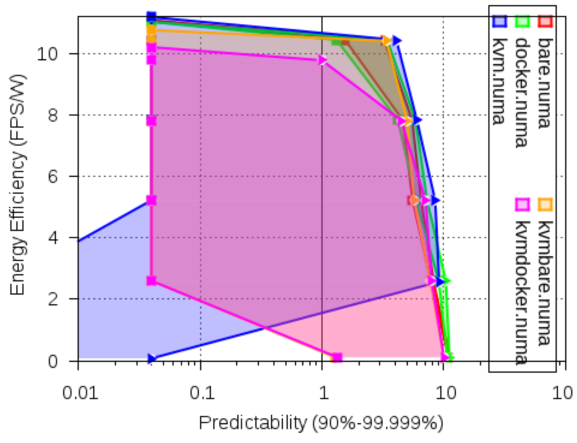


(a) Out-of-the-box deployment on host. (b) Application-optimized deployment on host.

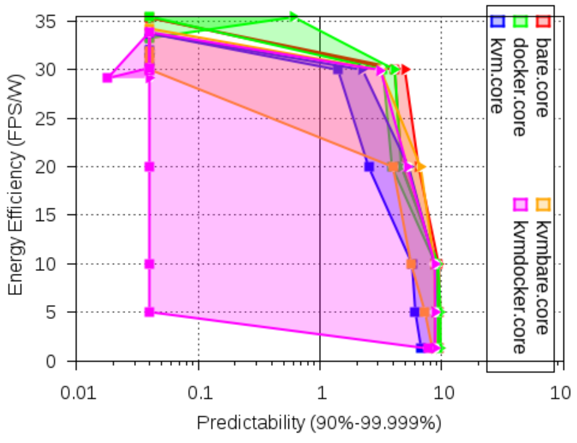
Figure 48 – Performance-Predictability Region for a YUV conversion benchmark with a target QoS range of 90-99.999%, in case of (a) an out-of-the-box and (b) an optimized deployment, in case of an Intel Atom Avoton 8-core server.

3.3.4.6 Other types of Performance Predictability Regions

Lastly, the performance results so far in these predictability curves and regions were absolute or relative aggregate throughput results. Other performance-related results, such as energy or cost efficiency can be used as well. One example is shown in Figure 49, where the energy efficiency (in FPS/Watt) is used as performance metric rather than raw FPS. In these graphs, one can see that the Atom micro-server is much more energy efficient in processing these frames than the classical AMD server, so in case energy efficiency (or cost efficiency) is most relevant, using the micro-server can be more beneficial for this application.



(a) AMD Opteron 24-core.



(b) Intel Atom Avoton 8-core.

Figure 49 – Energy-Predictability Region for a YUV conversion benchmark with a target QoS range of 90-99.999%, in case of (a) an AMD Opteron 24-core and (b) Intel Atom Avoton 8-core server.

3.3.4.7 Conclusions

In case of demanding and sensitive applications, the predictability of a particular runtime environment in a dynamic heterogeneous cloud environment is at least as important as the average throughput and (cost)efficiency of that environment for that application. To be able to efficiently visualize, summarize and compare the performance and predictability characteristics of an environment, we introduced the concept of performance predictability curves and regions. We explained how this could be measured for a particular type of applications, and demonstrated how they can be used in a number of use cases and example environments. From this, a clear conclusion that could be drawn is that predictability of a particular runtime environment for a particular application is very sensitive to various configuration settings (e.g., kernel version, deployment parameters, virtualization type, etc.).

3.3.5 Application QoE Monitoring

Especially in case of demanding and sensitive real-time applications, application services could provide multiple QoE metrics towards the heterogeneous cloud platform, so that the latter can better optimize how these services are being deployed and managed on particular cloud nodes. Especially in a cloud environment, where applications typically are sharing the same physical resources with other (possibly also demanding and sensitive) applications, enabling this feedback loop could be very beneficial for both entities: the service providers can deploy its services more cost-effectively and with higher QoE, whereas the cloud provider can provide a better quality of service and obtain a higher density with its infrastructure. Indeed, if a cloud provider can measure the impact of coscheduling particular services, it can either try to avoid this interference or try to mitigate this interference by enabling a stronger performance isolation.

Monitoring applications as a black box however only allows cloud providers to measure how effectively its resources are being used. It remains difficult or even impossible however how this resource behaviour is being perceived by the application itself. For example, the QoS provided by a cloud platform can be perceived very differently from a QoE perspective in case of an offline versus online video transcoding service: an offline transcoder will be mainly interested in the average throughput (and its cost-efficiency), whereas the online transcoder also has to worry about deadline misses, jitter, etc.

One minimal and essential application QoE metric that was proposed early on in the FUSION project is the session slots concept. This will be discussed in more detail in the following section. Session slots provide a coarse-grained view of the application QoE w.r.t. the number of sessions a particular application can support in a particular runtime environment with a particular minimum QoE. It however does not provide insights in how good this QoE level actually is. Additional and/or more refined application-specific metrics could provide this additional insight.

In this project, we provided the first step towards building such QoE-enabled platform by creating a QoE monitoring service where application service instances can log QoE metrics. We also built a dashboard UI to visualize these QoE metrics and allow comparison between application instances deployed on different platforms. This will be used as part of an evaluation scenario described in Deliverable D5.3. An example is depicted in Figure 50, where we visualize the key application QoE metrics for three video decoder session slots coming from three video decoder service instances deployed in a different manner on different hardware environments. For more details, we refer to Deliverable D5.3. In future work, these metrics could be integrated into the cloud platform for building a fully automated self-optimizing heterogeneous cloud platform.

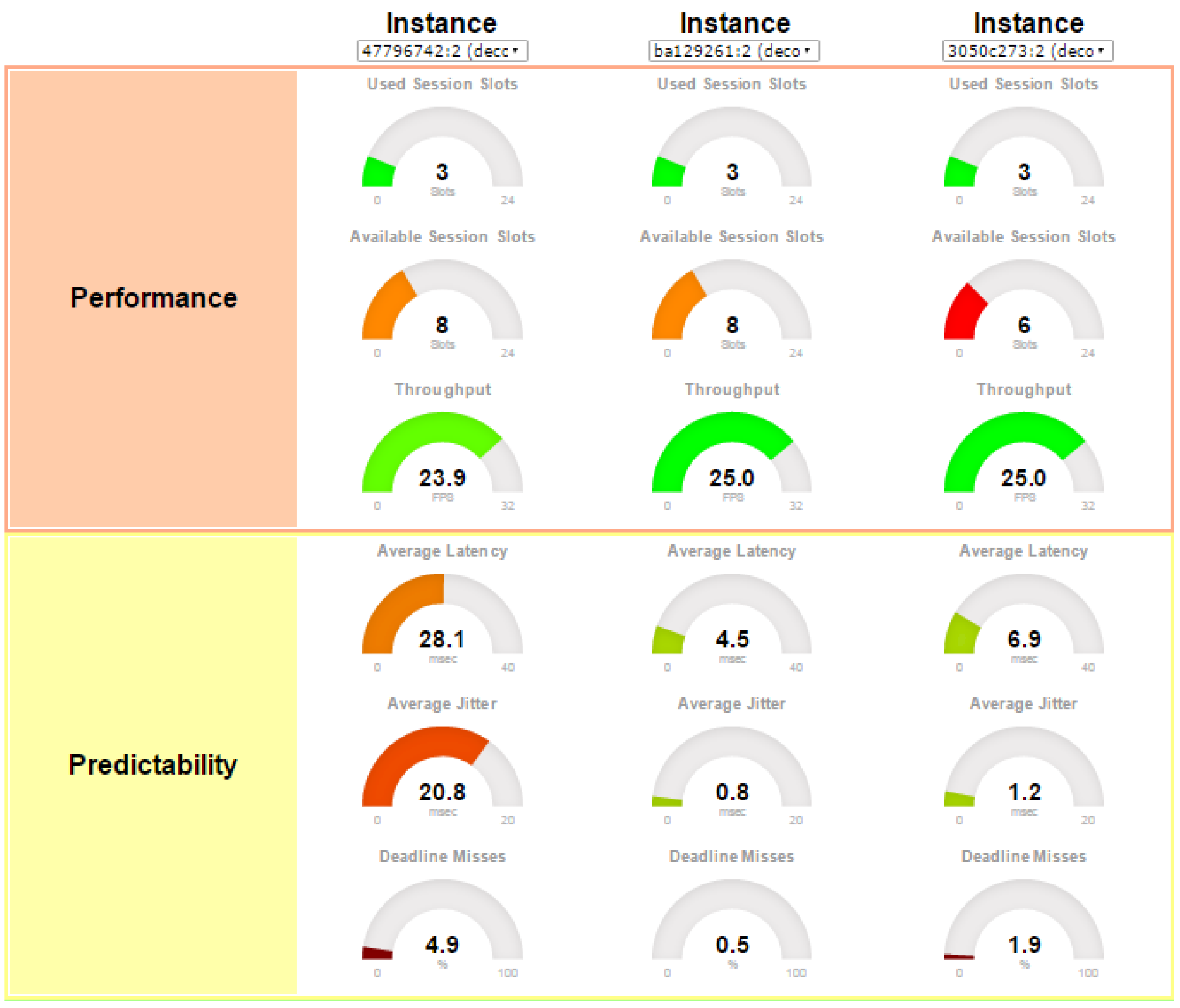


Figure 50 – QoE metrics dashboard.

3.3.6 Session slots

This section discusses and analyzes service session slots as lightweight abstract application monitoring metric for monitoring application services that are deployed in a distributed heterogeneous cloud environment.

3.3.6.1 Motivation

Typical monitoring metrics that are being used for measuring application resource utilization include metrics such as CPU and networking, but also application specific load metrics. Additionally, application specific high watermark (HWM) values are often used to indicate when a particular application VM is about to be saturated and when there is a need to scale out new application VM instances. A cloud load balancer, possibly application specific, is then used to distribute the incoming requests over the various application instances using some heuristic(s).

In a distributed heterogeneous cloud environment with long-lived demanding services such as FUSION, where each service session consumes a particular amount of resources for a particular amount of time on a particular runtime environment with particular hardware resources, accelerators and runtime execution characteristics, we need an efficient metric for monitoring available and remaining service capacity when deployed on a particular runtime environment. This metric will not only be used internally inside a zone (e.g. for intra-zone scaling or load balancing), this will more importantly be used as a lightweight metric towards the FUSION resolvers, so that they can easily decide on available application capacity in some execution zone without needing to have the detailed application and resource information as well as the application-specific HWM values per application and resource type.

3.3.6.2 Basic Concept

Already in the first year of the FUSION project, we came up with the concept of an application session slot as an lightweight metric to abstract the number of resources one user session requires for a particular application service on a particular runtime environment. Concretely, an application service announcing one available session slot by definition means that there are sufficient resources available to host one user session with *sufficient* QoE. What resources are required and what runtime characteristics a particular runtime environment needs to have in order to meet these criteria of available resources with sufficient QoE is very application dependent and as such this complexity should not be necessarily exposed externally.

In general, a service instance announcing *N* available session slots by definition means that this instance has sufficient available resources to host an additional *N* parallel user sessions with sufficient QoE. For a service instance to be able to announce his availability, we developed specific API calls either the zone manager and/or application service can implement to push and/or pull this information so that the FUSION cloud platform can process this monitoring information, for example by updating the information in the FUSION resolvers.

In FUSION Deliverables D3.1 and D3.2, this basic concept, its benefits and basic extensions were already discussed and qualitatively evaluated in detail, so we will not repeat this in this deliverable, but only provide a short overview of the key properties and benefits.

In summary, the key benefits of session slots, include:

- Light-weight service request resolution
- Separation of resource allocation and service request resolution
- Easy hierarchical aggregation of service availability
- Service type neutrality
- Stability

- Lightweight auto-scaling
- Easy billing
- Trust

Some key drawbacks of this concept, include the following:

- Application-specific (internal) monitoring and mapping onto a realistic session slot availability.

  Note that this could be left to the evaluator services, using a conservative number that is used as a basis for that environment, rather than continuously trying to monitor and adjust the available session slot metric.

- Wildly variable resource consumption

  In case the required resource consumption can differ substantially over time, it may be difficult to provide a reasonable estimate of the true available session slots, and as such may require a very dynamic internal monitoring and correction mechanism, which makes these lightweight application services more complex. Note that this could be partially mitigated in case one of the service components in the graph is responsible for monitoring and adjusting session slot information.

In Deliverable D3.2, the basic concept was applied in a number of additional use cases:

- Efficient inter-zone and intra-zone service scaling, using available session slots as a metric to decide whether to scale in or out.
- Session-slot based intra-zone load balancing, using session slot availability across various instances to balance the load.
- Service aliasing, separating resource utilization from service utilization, allowing multiple related services to reuse the same resource session slots from the same basic instances.
- In heterogeneous cloud optimizations, using session slots as a basic metric for summarizing and comparing service resource utilization efficiency.

3.3.6.3 Extended Concept

As discussed in Section 0, it is not always feasible to pre-deploy instances of all services in appropriate execution zones. In such cases, we need to fall back to on-demand service placement and deployment scenarios, where services can be quickly deployed on-the-spot in an appropriate execution zone. For different execution zones, the amount of time this takes, can vary significantly.

To avoid always needing to include the domain orchestrator for on-demand service placement, an alternative approach is that each execution zone for particular services also provides a metric related to the amount of time a user may have to wait on average in case no slots would be readily available. Remember that the session slot availability is basically an indication of the amount of users that can be handled immediately by some service instance without having to wait for previous users to be handled.

The response times of classical request-response services such as web services or databases can typically be modelled with classical queuing theory, where incoming requests are queued and dispatched across the available servers (or worker threads). In this model, the number of session slots represents the number of servers or worker threads that can process incoming requests in parallel. Due to the short-lived nature of these processing requests however, the queuing behaviour typically dominates for these types of services.

For long-lived FUSION services however, the session time or processing time will typically dominate as sessions will typically last for several minutes to hours, and as such the default queuing does not make a lot of sense here, as nobody will wait for minutes to start using some service.

However, this concept of queuing and queuing delay could however be used to accelerate the on-demand scenario so that a FUSION resolver does not always have to contact the domain orchestrator but can intelligently decide itself which execution zone to contact directly.

Concretely, in case execution zones would have the flexibility to autonomously deploy new instances of a particular service (even though that service may not even be running in that zone at all yet), then apart from the session slot availability, a zone could also announce the average queuing time for such service. The average queuing time would then represent the expected amount of time it would take for that execution zone to enable a new session slot for that service. For example, in case the queuing time is 3 seconds, this means that zone could spawn new instances that are fully up and running within 3 seconds on average. Note that the on-demand provisioning algorithm presented in 0 combined with the lightweight application service model, this average bootstrapping time can be reduced significantly.

This queuing time is expected to be quite variable across execution zones, but also within the same execution zone (e.g., due to different start-up behaviour over time). For example, some zones may not support or implement this fast-track on-demand deployment scenario, or may not (want to or be able to) support this for all services. This could be modelled using a (default) infinite queuing time for those services and/or execution zones. Others may have implemented the fast on-demand provisioning algorithm whereas others use a more lazy approach. Some zones may have dedicated storage, networking and runtime/OS capabilities to accelerate container deployment, whereas others may not. The queuing time may also vary over time, for example due to resource constraints. Some zones may only be able to spawn a limited or maximum number of on-demand instances in parallel, whereas others may not have such stringent limitations.

In general, the combination of session slots and a queue delay is depicted graphically in Figure 51. In case there are still available session slots, then a user can immediately consume that slot and connect to the application service without delay. If there happen to be no slots available, then, in case there is a finite queuing time defined, the incoming user request is queued and a new instance is deployed on-demand. When the instance is up-and-running, the user is granted a session slot.

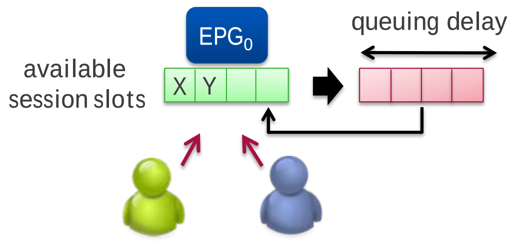


Figure 51 – Extended model of session slots and queuing delay.

This queuing delay (typically but not exclusively referring to the on-demand deployment delay) is a feature controlled and managed by the zone manager. A service however could indicate the maximum tolerable queuing delay. Services using an on-demand scenario however themselves should also optimize themselves to minimize the application start-up delay, for example by already accepting and announcing slots immediately after start-up, while the remaining libraries, data and/or background services are being started or fetched.

A zone manager supporting this fast-track on-demand deployment scenario announces for each supported service the respective expected average queuing delay to the FUSION resolvers. Even if there are no instances (and thus zero session slots) running in a zone, the zone manager can still announce the availability of that service with zero session slots and the corresponding queuing delay.

A FUSION resolver can use this information for speeding up the on-demand service deployment case. In this fast-track mode, in case there are zero useful session slots available, rather than contacting the domain orchestrator, the resolver can leverage this queuing delay information to decide which execution zone to contact directly. Note that in case a zone uses a service load balancer with a corresponding IP address, the resolver can already immediately return the corresponding IP address to the requesting client. In parallel, to speed up the on-demand deployment, the resolver can trigger the zone to start deploying a new instance of that service. In this fast-track mode, the orchestrator is not (directly) involved, significantly improving scalability and reducing overall start-up latency.

3.3.6.4 Analysis

One of the key benefits of session slots as a capacity monitoring metric is that all complexity for calculating the remaining discrete capacity is tackled near the source (e.g., inside the service or via a helper service), so that all remaining components (e.g., zone manager, resolver and domain orchestrator) can efficiently process this metric, rather than having to rely on one or more continuous metrics (e.g., CPU and networking usage) and needing to know how to translate this in effective service capacity and compare with other execution zones.

Two additional benefits are that they are very easily to aggregate within and across zones simply by adding up all values from all instances of that service in a zone and across zones in a geographical region. Moreover, for a resolver, only an order of magnitude is really relevant rather than an exact number, so only changes in the order of magnitude of available session slots could be reported to the resolver to reduce the monitoring data to be exchanged. For example, the relative difference between one or two available session slots is much more significant than the difference between 100 and 101 slots.

In general the overhead for sending service monitoring information between execution zones and resolvers can be modelled as follows:

$$R = M \times U \times Z \times B$$

With M being the average message size of one single update, U the total number of service updates per second per zone, Z the total number of zones in a domain and B the number of resolvers each zone directly sends monitoring updates to. In case of metrics such as session slots and queuing delays, M typically will be only a few bytes (e.g., 4 bytes), by using varying encoding and compression mechanisms for identifying the service ID, the session slot number and queuing delay. In case for example only the $log_a(\#session\ slots)$ is exchanged, then only a few bits suffices (obviously with reduced accuracy and aggregatability). For example, one could use 4 bits and a log_2 scale to represent 16 bucket levels (0, 1, 2-3, 4-7, 8-15, ...) of session slots. In the extreme case, one could even decide to only use 2 bits and e.g. a log_{10} scale to represent only 4 categories of session slot availability: 0, 1-10, 10-100, and 100+. Note that one could also decide to send the actual session slot number (e.g. using 8 bits), but only send them when there is an order of magnitude difference with the previous message exchange. In that way, one preserves the aggregatability capability at the resolvers at the expense of some more bits per exchange.

In case the queuing delay is also transmitted, also there one could decide to only exchange orders of magnitude differences in queuing time, and only exchange this information when there is an order of magnitude difference in queuing time. For example, one could represent queuing time as follows: $Q = 0.1 \times 2^q$, using only a handful of bits to represent values for q (e.g., 4 bits). In both cases, this is a trade-off between accuracy and overhead, though will limited expected impact on the performance of the resolution process.

In the simple formula earlier, Z is the number of zones in a domain. For this initial analysis, we will assume Z=10,000. Each zone can directly forward its session slot updates to a number of resolvers. Especially in case only log-values with limited aggregatability are being used, it may be beneficial to exchange this information with a number of resolvers. In this analysis, we will assume B=1, so each zone only directly exchanges information with its prime resolver. Under these conditions, the simple formula then reduces to:

$$R = 40KB \times U$$

Where U is the total number of updates per second across all relevant services in that one. In case $U = 100/s$, then $R \approx 4MB/s$. On the other hand, if $U = 10{,}000/s$, then $R \approx 4GB/s$. Note that in case other metrics would be used, one could expect that the constant factor *M* would be higher, as well as the value for *U,* as it is expected that more frequent updates may need to be exchanged.

The key parameter as such is the rate with which session slot updates need to be exchanged. As mentioned earlier, it is expected to be sufficient to only exchange differences in orders of magnitude, and only at a maximum frequency rate. Intuitively, the probability that there is a difference in order of magnitude *a*, increases as the number of available session slots goes to zero, as is shown in this formula:

$$P = \frac{1}{a^{p+1} - a^{p}} = \frac{1}{a^{p}(a-1)} \approx \frac{1}{a^{p}}$$

In this formula, a^p represents the amount of session slots as an order of magnitude. Thus, the closer p gets to 0, the higher the probability of a possible update. In a demand-driven cloud environment however, one typically want to minimize the amount of available session slots to reduce the cost overhead. As such, it is expected for p to be relatively small in most cases.

A more detailed analysis is needed to investigate the impact of the frequency of updates per zone with the performance of service resolution. A crucial factor to investigate is related to the amount of times new instances need to be spawned on-demand in a zone that did not have any available slots anymore, even though there were actually still available slots in other nearby execution zones; or vice versa, the amount of times the resolver assumed no available slots in a zone, where in reality there was at least one, possibly resulting in a slightly worse utility function.

3.3.6.5 Conclusions

One of the key FUSION concept is the concept of a session slot as an abstract application monitoring metric to express the remaining available resources for a particular application service instance in a particular heterogeneous runtime environment. We extended the concept presented in Deliverable D3.2 even further by introducing the additional concept of a queuing delay next to session slot availability. Especially in case of lazy service deployment and on-demand service deployment, this metric can give the service resolution plane crucial information on how long it would take to create additional session slots of a service in an execution zone. We subsequently also analyzed the benefits of session slots in terms of reduced monitoring data that needs to be exchanged between a zone and resolver, and presented mechanisms for further reducing the total bandwidth.

3.3.7 Evaluator Services

3.3.7.1 Motivation

Optimally deploying applications with very stringent requirements (as FUSION is targeting) onto distributed resource-constrained heterogeneous cloud infrastructure is a complex problem. First, a service may have specific hardware and software resource or performance requirements to deliver consistent QoE towards all end users. Second, the resource capabilities as well as perceived performance/QoS may vary significantly in such heterogeneous environment. Third, different environments across different data centres may vary significantly in price, and service providers may want to decide on how much they are willing to pay for a particular QoS provided by some pool of (virtualized) resources.

Specifying all these requirements and trade-offs in static manifests, extracting detailed static and runtime knowledge of the available hardware resources, and finally combining all this information to be able to infer feasible and optimal deployment locations, would result in a **very complex system** (e.g., a rule engine) that needs to be able to *understand* all requirements, capabilities and their corresponding relationships. Moreover, it would still be **incomplete**, as new applications may have different requirements that cannot be captured, processed or understood by current available system. It would also require to explicitly identify and expose all application requirements or resource capabilities and constraints, which can prove to be difficult, complex, or may result in unacceptable overhead, or is practically impossible due to intellectual property concerns.

3.3.7.2 Basic Concept

To efficiently tackle the problem summarized in the previous section, we introduced the basic concept of an evaluator service in FUSION already early in the project (see D3.1 and D3.2). To recap, an evaluator service basically is an active service probe that is (deployed and) triggered prior to service deployment, and that evaluates a particular (virtual) runtime environment by generating a **score**, comprising all application-specific functional tests and trade-offs. These scores can subsequently be used as key input for building efficient optimal service placement algorithms.

As a service provider will be charged for running the service, he should be able to decide what he is willing to pay for. Rather than trying to capture all detailed service requirements and placement policy trade-offs in a complex static manifest (thereby also requiring the orchestration layer to be able to interpret and cope with these complex manifests), the core idea is to offload this complexity into active probes that evaluate the feasibility and/or cost-effectiveness of running a particular service in a particular execution environment and return a score to grade the evaluation. These probes may be generic, resource specific or service specific.

For example, services may require specific hardware or a certain proximity to other service instances or have different runtime requirements towards the execution environment. For instance, a real-time rendering service may depend on a GPU, or certain GPU capabilities, and a corresponding streaming service nearby. These requirements may include specific OpenGL extensions, a specific minimum OpenCL version or supporting specific OpenCL extensions, or vendor specific hardware and APIs such as NVIDIA CUDA support. Effectively, most 3D games have particular minimal GPU requirements. Additionally, a service provider may offer specific quality modes for a particular game, which translate into more specific hardware requirements. For example, certain realistic real-time lighting and shadowing techniques depend on the availability of a geometry shader.

Describing these kinds of restrictions in static manifest files would result in requiring the orchestrator to understand all possible use cases of all applications. The descriptions would also need to be updated whenever new hardware or hardware revisions become available. Particular runtime environments would also need to be characterized to capture and expose their runtime behaviour.

Therefore, a static approach using detailed manifests quickly becomes difficult to manage for large numbers of services and therefore unscalable (though it could be sufficient for an architecture designed for a specific and limited subset of services with a priori known requirements). Instead, we propose using evaluator services that are deployed on the actual environments as active probes. These evaluator services can be provided by the application service provider (or possibly an evaluator service provider), allowing application and service provider specific checks and cost-utility trade-offs to be made by the service provider. They are deployed within various execution environments of several EZs (based on the service provider policies) and are automatically triggered by the domain orchestrator placement function as part of service placement for determining the optimal location(s) for deploying new instances of a particular (set of) service(s).

3.3.7.3 Design Considerations

The three main design considerations are simplicity, flexibility and efficiency. **Simplicity** means that it should be simple to leverage evaluation scores optimizing service placement. As such, we envision these scores to be as basic as a float or integer, abstracting the complex trade-off between static requirements, runtime behaviour, QoS and cost, rather than using complex scores containing multiple values representing different aspects (cost, efficiency, etc.).

The second design decision is **flexibility**. Service providers should be in the loop when deciding where their services should be deployed, a feature that is naturally supported by evaluator services. On the other hand, a domain orchestrator or zone manager also may want to enforce some policies or reserve some (compute or networking) resources for more profitable services. As such, we allow for a dynamic pricing model where the price of execution environments can change based on changing policies or changing runtime behaviour, allowing domains and zones to steer the decisions of the evaluator services by dynamically changing the price. The cost of running a service in a particular environment is one of the key input parameters of an evaluator service, allowing the service provider to return a proper score with respect to the cost. Note that this score can also change over time due to changing policies of the service provider. As such, we envision a score to be typically only valid for a limited amount of time.

The third design decision is **efficiency**. Three key factors are the overall response time and the deployment and runtime overhead. As such, evaluator services should be optimized for quickly returning a score upon an evaluation request. For some evaluators, this could involve doing part of the evaluation as a background process. Also, as evaluator services may need to be deployed just-in-time in remote data centres, the provisioning and deployment time should be minimal. Hence, a good candidate for quickly provisioning and deploying new instances in particular locations are lightweight containers such as Docker. The issue of runtime overhead is discussed further in Section 3.3.7.7.

3.3.7.4 Basic Model

In general, an evaluator service is a function that, given a set of input parameters (including the environment, historical data, policies, etc.), returns a value that can be considered as a score or rank, indicating how suitable that environment is for deploying a number of session slots of a particular service:

score = evaluator(Service,InstParams,Env,Cost(Env,t))

Service represents the service as well as its requirements, *InstParams* the instantiation and configuration parameters for deploying that service (e.g., UHD quality, premium QoS, etc.), *Env* represents the runtime environment, and *Cost(Env, t)* represents the cost of that environment in a given time frame (in general, this is a combination of the absolute time and the duration).

A minimal property is that the resulting scores should be (partially) **ordered** for a particular service type: a (slightly) preferred environment should have a (slightly) higher score, allowing to simply order all tested environments based on their score.

In the context of global multi-service placement algorithms, there can also be additional benefits of associating additional properties to these scores, such as proportionality (i.e., an environment that is twice as good results in a score that is twice as high), or allow for a more specific interpretation of the scores (e.g. as a bidding value in an auctioning placement algorithm or an average execution time or runtime latency).

3.3.7.5 Extended Model

In the initial basic model, an evaluator service is an active service probe running on some runtime environment in some remote execution zone. During the analysis of the benefits and overhead of the evaluator service concept, it became clear very quickly that we needed to refine this concept to reduce the overall overhead while at the same time improving its overall flexibility and effectiveness.

As such, we refined the evaluator service model and APIs in the final year of the project by splitting the functionality and role of an evaluator service in two key stages:

- **Stage 1 evaluation probe(s)**

  This basically is a pure runtime environment probe that performs a capability and capacity analysis of a particular runtime environment for a particular (set of) application service(s). As an example, this probe could return the estimated number of supported session slots, though the returned value or structure could be of any type or semantic. Multiple probes, assessing different aspects (e.g., networking, compute, etc.) could potentially be deployed in parallel, evaluating different aspects (throughput, latency, jitter, etc.).

  More concrete, these probes can be used to evaluate all *unique* runtime environment types w.r.t. both the local compute/network/storage capabilities as well as its global network capabilities and geographical location. So, in typical cases, there will be typically *one-per-unique* environment type that typically runs *once-per-unique* environment type. In many cases, evaluating a particular unique environment once during its lifetime can be sufficient, and does not have to be repeated during every domain placement decision, drastically improving scalability and reducing runtime/cost overhead. However, other scenarios, such as a periodic trigger is also possible, for example in case of active profiling, where an evaluator service wants to build a statistical profile of the long-term runtime capabilities and behaviour of a particular environment.

- **Stage 2 evaluation analysis**

  This part of the evaluator is responsible for doing a cost-benefit analysis of deploying a service in a particular runtime environment. These evaluators typically interpret and combine the results from the various evaluator probes and combine this with the latest costs, policies and other parameters for finally ranking the different environments for a specific deployment request.

  In typical cases, there is only *one-per-domain* of these analyzers necessary per (group of) application services, which can be collocated closely to the domain orchestrator, and these typically will be triggered *once-per-unique* placement evaluation. For example, these cost-benefit evaluations do not have to be repeated if the environments, their relative prices, and the minimum required available environment instances remained constant since last placement request.

  Note that the result here could be more than only a simple ranking or scoring of the various environments. As in [NAA10], an stage 2 analyzer could provide multiple *Q-scores*, representing multiple QoS or performance levels, each having an associated cost. These Q-scores represent the willingness to pay some amount for a particular QoS or performance. An example is depicted in Figure 52. It is then up to the domain orchestrator to ensure that all services at least achieve their minimal Q0 level, but try to maximize the Q-levels for particular services. A huge advantage of having these stage 2 analyzers is that these Q-levels can easily be dynamically change over time based on the policies of the application service provider.

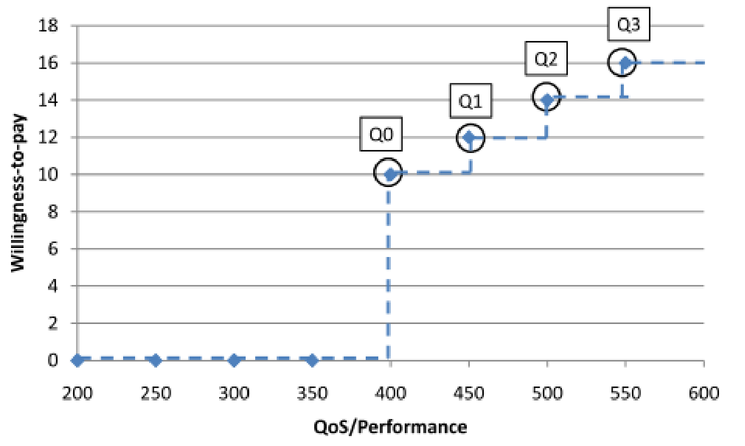


Figure 5. Example of application QoS and definition of Q-states and willingness-to-pay values.

Figure 52 – Example graph illustrating the concept of Q-scores (taken from [NAA10]) as a trade-off between QoS/performance and the willingness to pay for such QoS/performance.

Note that in both cases, these evaluator services could be either application service-specific, application service class specific, or general purpose. In the latter cases, the same evaluator (probe or analyzer) could be reused for a larger group of service types, providing a trade-off between reusability, specialization and overhead.

With respect to the nature of both evaluator services, we also can distinguish between different types, trading off (cost) overhead with flexibility and programmability:

- **Static Manifest file / formula**

  This is the option that is typically used in cloud orchestration frameworks, and is also provided for backward compatibility, as this option can be sufficient for services without stringent requirements. Basically, a static manifest (stage 1) or formula (stage 2) describes all static application and runtime requirements, and it is up to the domain and zones to evaluate these requirements or evaluate the formula. This option has the benefit that no additional actively running components need to be developed or managed, at the expense of limited flexibility.

- **Passive script**

  In this model, the stage 1 or 2 evaluator service consists of a passive script (e.g., a Python script) that is interpreted in a sandboxed environment by the domain orchestrator or zone manager. This allows for improved flexibility while still avoiding application-specific actively running components to be developed and managed. However, the number of tests such passive script could do, is still limited. One could not run a particular benchmark, trigger some GPU capabilities, etc.

  For many stage 2 evaluator services, this model could be largely sufficient.

- **Active FUSION service**

  In this model, there is an actual FUSION service deployed to do the evaluation. In this model, any implementation could be used to do the assessment (e.g., run the actual service in some performance mode), resulting in the highest accuracy and flexibility, though at the expense of having to develop and manage for these additional running services in a FUSION environment.

  For application services with stringent requirements, this model may be necessary.

Finally, an updated sequence diagram of w.r.t. evaluator service based placement in case of stage 1 and stage 2 evaluators, is depicted in Figure 53:

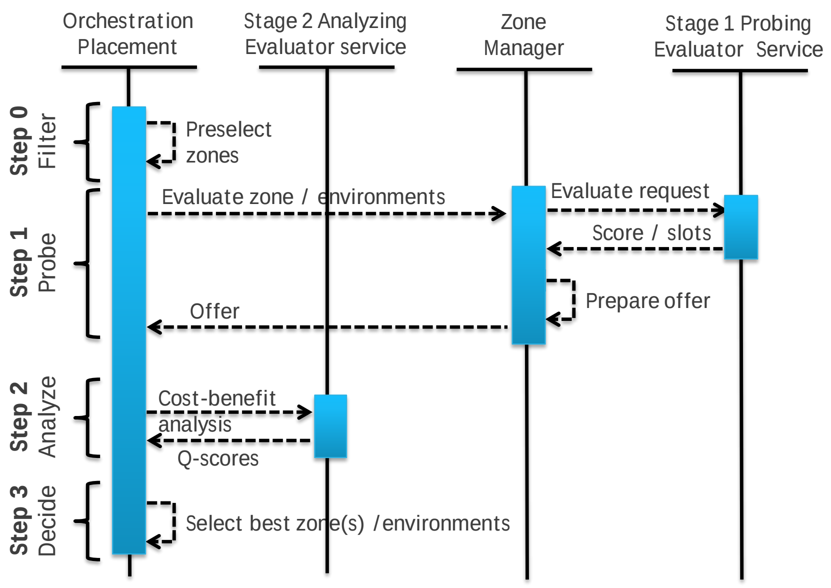


Figure 53 – Updated sequence diagram of service placement with stage 1 and 2 evaluator services.

In this updated model, placement consists of up to four key steps, namely filtering, probing, analyzing and deciding:

- **Step 0: Filtering**

  In this step, the domain placement function will filter out all irrelevant execution zones to be considered for a particular service placement request. This pre-filtering could be done based on any type of information, such as the location of the zone, the cost, etc.

- **Step 1: Probing**

  For new zones or new runtime environments, stage 1 evaluator probes will be deployed and/or triggered to evaluate the capabilities of these environments.

- **Step 2: Analyzing**

  In this step, the domain placement function will trigger the stage 2 cost-benefit analyzer evaluator service to analyze and rank the various environments w.r.t. each other, based on various parameters such as costs and policies.

- **Step 3: Deciding**

  The domain placement function in this last step will decide on the (set of) locations where to deploy one or more instances of a particular service, leveraging the results from the previous step.

In the following sections, we will not further discuss, analyze and evaluate the key benefits and overhead of having these stage 1 and stage 2 evaluator services.

3.3.7.6 Key Benefits

Evaluators will identify the feasible EZs in terms of their capabilities and efficiency, while service placement will select between them for optimizing the utility function as discussed in Section 4.4 of Deliverable D3.2. Prior service evaluation will ensure that the selected EZs for placing service instances have the required software and hardware capabilities, which will significantly reduce the total number of EZs under consideration for the placement optimization algorithm, improving scalability and performance. Secondly, no detailed information of the application specifications or available EZs resources needs to be disclosed with the central orchestrator to be able to do optimal service placement.

3.3.7.7 Analysis

3.3.7.7.1 Definitions

In this section, we analyze and discuss the trade-off between the cost and runtime overhead of running these stage 1 and 2 evaluator services w.r.t. their benefits. The **runtime overhead** is defined as the relative runtime overhead for running these evaluators compared to the total run time of all service instances that have been deployed using these evaluators. The **cost overhead** then is the relative cost of running these evaluators compared to the aggregated total cost of running the actual service instances.

W.r.t. the benefits, we only consider the **cost benefit** of being able to either support a higher number of available session slots for a particular **unit cost** (i.e., cost per slot per hour), or for being able to support the same number of session slots for a lower unit cost.

Note that in this model, we do not consider any QoS/QoE benefits, as by definition, one session slot is equivalent to the set of runtime resources and behaviour for supporting one user session with a particular minimum QoS/QoE level. Moreover, in our current model, different QoE levels of some service are supported by FUSION through different service names, which can be mapped onto the same physical instances through our multi-service configuration concept. For example, a gaming service provider could provide two QoE levels for a particular game, through a basic and premium account. In FUSION, this would map onto two different service names (e.g., *mygame.basic.fusion* and *mygame.premium.fusion*), but can be mapped onto the same physical service instance. It is then up to the instance to provide the correct mapping of session slots (e.g., 1 premium slot = 3 basic slots).

3.3.7.7.2 Problem Space

Before presenting and discussing the overhead models, we first investigate the problem space itself. In worst case, a stage 1 evaluator service probe could be deployed in every single *runtime environment type* across all available execution zones. A runtime environment type is a particular runtime environment that has particular runtime capabilities, properties as well as particular runtime behaviour. Example capabilities and properties include the number and type of CPU cores, the available memory, networking, storage, etc. Example runtime behaviours include how a particular runtime environment behaves over time, both w.r.t. the average behaviour, jitter and tail behaviour. Especially in cloud infrastructures, the amount of jitter and tail behaviour can be significantly higher than on a bare environment that has exactly the same capabilities. Evaluator probes need to be aware of this when probing some cloud runtime environment.

Consider the following set definitions:

$$Z = \bigcup_{i} Z_i = \{available\ zones\ Z_i\}$$

$$Z' = \bigcup_{i \in F} Z_i = \{subset\ of\ available\ zones\ to\ be\ considered\ (i.e., zone\ filtering\ in\ step\ 0)\}$$

$$V_i = \bigcup_{j} V_{i,j} = \{available\ environment\ types\ V_{i,j}\ in\ Zone\ Z_i\}$$

$$V_i' = \bigcup_{j \in f} V_{i,j} = \{available\ environment\ types\ in\ Zone\ Z_i\}$$

$$V_i'' = \bigcup_{i \in F, j \in f} V_{i,j} = \{subset\ of\ environment\ types\ to\ be\ considered\ in\ selected\ Zones\ Z_i'\}$$

$$V_u = \bigcup_{i}^{Z} V_i = \{available\ \boldsymbol{globally\ unique}\ environment\ types\ across\ all\ zones\ Z_i\}$$

$$V_u' = \bigcup_{i}^{Z'} V_i = \{available\ \boldsymbol{globally\ unique}\ \ environment\ types\ in\ selected\ zones\ Z_i'\}$$
$$V_u'' = \bigcup_{i}^{Z'} V_i'\ = \{subset\ of\ \boldsymbol{globally\ unique}\ \ environment\ types\ in\ selected\ zones\ Z_i'\}$$
$$V = \bigcup_{i}^{Z} V_i = \{available\ \boldsymbol{locally\ unique}\ environment\ types\ across\ all\ zones\ Z_i\}$$

In the equations above, the difference between V_u and V is the ability to identify identical runtime environments across different zones, e.g. by means of a unique ID. In case zones are operated by the same zone provider, this can easily be done. In case zones are operated by different providers or entities, this can only be achieved in case there is some standardization in the definition and management of runtime environments.

During service placement, and thus also evaluator service placement, typically only a subset of all available execution zones and only a subset of all available runtime environments need to be considered. The filtering of zones typically is done based on global parameters such as network capabilities and geolocation; the filtering of runtime environments is done on minimal static requirements that should be at least fulfilled. For example, it makes little sense to deploy an evaluator probe for a GPU rendering service on a runtime environment without a GPU, with not sufficient CPU cores or memory. This filtering can drastically reduce the number of environments to include in the placement decision (at the different orchestration layers), thus also reducing the overall overhead and improving scalability. Note that this filtering could be done in an adaptive smart manner, learning from other evaluation results to more aggressively branch-and-bound related environments.

Another key insight with these definitions is the importance of identifying and unifying which environment types are identical across execution zones, and which environments are globally unique, as this will also drastically reduce the total number of environments to evaluate.

Imagine for example that $|V_u| \ll |V|$, then $N \approx \max(|V_u|, |Z|)$, with N being the total number of environments to be deploy evaluator services on (we currently assume no pre-filtering). Indeed, to be able to assess the local runtime characteristics of the various runtime environments, we only need to deploy one evaluator probe per globally unique environment, irrespective of the execution zone. However, in case we also want to assess the global characteristics (i.e., geolocation & global networking), we also minimally need to deploy at least one probe (e.g., a networking probe) per execution zone. In case $|V_u| < |Z|$, then we first intelligently select unique environment types across as many different execution zones as possible, and only need to deploy additional probes in the zones that do not have any probe yet. So we ideally want only one probe per zone but still assess all unique environment types at the same time. In case $|V_u| > |Z|$, then we also intelligently select unique environment types across all available zones, so that we do not have to deploy additional probes only to cover all execution zones. Note that in case of filtering, the same strategy can be applied: if $|V_u''| \ll |V''|$, then $N'' \approx \max(|V_u''|, |Z'|)$.

For example, assume a single cloud provider such as Amazon, which currently has roughly 40 instance types, corresponding to unique runtime environment types, mostly shared across almost all regions. They currently have about 10 regions, each with on average about 3 availability zones or DCs (i.e., execution zones). Consequently, $|V_u| \approx 40$, $|Z| \approx 3 \times 10$ and $|V| \approx 1200$, so in this case the total number of unique environments to be evaluated across all Amazon execution zones is only about 40 if we would smartly deploy all evaluator probes across all global Amazon DCs (only trying one or two EC2 instance types per availability zone), instead of 1200 if we would treat each zone independently and as such individually want to evaluate all global environment types.

3.3.7.7.3 Stage 1 Overhead

Let us first look at the overhead for a stage 1 evaluator. Assume the following parameters:

$N_i = \#\, deployed\ stage\ i\ evaluator\ probes, with\ i\ \in \{0,1\}$
$E_i = fraction\ of\ time\ one\ evaluator\ is\ running$
$I_A = average\ number\ of\ instances\ of\ application\ service\ A\ that\ are\ constantly\ running$
$S_i = evaluator\ service\ sharing\ factor\text{: } evaluator\ results\ can\ be\ shared\ by\ S_i\ service\ types$

Then the stage 1 runtime overhead can be modelled as follows:

$$R_1 = \frac{N_1 \times E_1}{I_A \times S_1}$$

The numerator represents the total aggregated fraction of time that all probes are running. For example, if 10 probes are deployed, each running about 10% of the time, then this amounts to one probe running all the time. In the denominator, the total application runtime is presented, multiplied by the number of service types S_1 across which this evaluator can be amortized: the larger the pool of actually running service instances (possibly coming from multiple (classes of) application services, the smaller the relative overhead.

The stage 1 cost overhead can then be modelled as follows:

$$C_1 = \rho_1 \cdot R_1 = \frac{\overline{Cost}_{probe}}{\overline{Cost}_{appservice}}_{weighed} \cdot R_1$$

In this approximating formula, we simply weigh the runtime overhead with the relative cost ratio ρ_1 for running such evaluator probe in the various runtime environments compared to the average weighed unit cost of running the actual application service(s) in their runtime environments. The unit cost of an evaluator probe could be multifold, depending on the pricing model used. In general, this unit probe cost is a function of the development cost of the probe and the actual runtime cost. For the runtime cost, one model could be for the domain orchestrator to only charge a flat fee per deployed probe, irrespective of the actual cost of running an evaluator in a particular runtime environment. In a very extreme case, the cost of running probes could even be zero (up until some threshold), in order to attract new customers or new services to particular runtime environments. Alternatively, the cost could also be a fraction of the actual cost of the runtime environment, or simply be as costly as running an actual application service in such runtime environment. Which cost model works best depends on the business model used, but obviously will strongly influence the overall cost overhead and corresponding deployment strategy used by application providers for deploying these probes (i.e., aggressively versus conservatively).

In this formula, the application service cost is the average weighed unit cost for running the application service(s) in their chosen runtime environments. This can be a mix of relatively cheap and/or expensive runtime environments (e.g., a central general purpose cloud environment versus a specialized GPU-enabled edge cloud environment).

Factor ρ_1 in this formula is the unit cost ratio of running an average evaluator service versus an application service. If $\rho_1 < 1$, then the unit cost of a probe is cheaper than the unit cost of an actual service. If on the other hand $\rho_1 > 1$, then the unit cost of a probe is more expensive than that of the actual service. Imagine for now that the cost of deploying a probe is as expensive as running an actual service (so no flat fee or fraction), and let's ignore the development cost for now and only consider the runtime cost overhead of a probe. In this case, ρ_1 depends on the distribution of the unit costs of all runtime environments **and** the corresponding service requirements.

For example, if the application service is a *lightweight* service with only mild runtime requirements, a relatively cheap environment will typically suffice. However, then $\rho_1 > 1$, as the probes in general will also be deployed on more expensive environments. In this case, a good optimization strategy is to keep ρ_1 as small as possible, and thus try to prune expensive environments (e.g., GPU edge node) with smart or explicit filtering techniques. A smart technique would be based on learning from the results from previous probe deployments, whereas an explicit technique would be to specify in the manifest how to explicitly prune environments with a particular unit cost or that have particular features.

On the other hand, in case of a *heavyweight* service with stringent runtime requirements, more expensive environments typically will be needed, and thus $\rho_1 < 1$, as in general many (too) cheap environments will be evaluated as well. Consequently, a second optimization strategy is to prune cheap environments in case of heavyweight services, again using smart and/or explicit environment filtering.

In general, a good cost optimization strategy is to avoid trying out too many suboptimal runtime environments, by smartly and/or explicitly filtering runtime environments that individually and/or combined are as close as possibly to $\rho_1 \approx 1$, even though the optimal relative unit cost of running a particular application service in a particular distributed heterogeneous FUSION cloud environment may not be known in advance, and thus need to be learned. Given this insight and objective, in the following paragraphs, we will assume $\rho_1 \approx 1$, and as such, we will only consider the runtime overhead in the remaining evaluation.

3.3.7.7.4 Stage 2 Overhead

The general formulas for the stage 2 evaluators basically are identical as for the stage 1 evaluator probes:

$$R_2 = \frac{N_2 \times E_2}{I_A \times S_2} \quad \text{and} \quad C_2 = \rho_2 \cdot R_2 = \frac{\overline{Cost}_{eval}}{\overline{Cost}_{appservice}^{weighed}} \cdot R_2.$$

However, for the stage 2 analyzers, typically $N_2 \approx 1$ (1 analyzer typically suffices), $S_2 \gg 1$ (1 analyzer per service class or app provider), and $\rho_2 \ll 1$ (stage 2 analyzer can run in cheap lightweight central runtime environment near domain orchestrator, and may be 0 in case of passive analyzers that are executed by the orchestrator). As such, in typical cases, $C_2 \approx 0$, and as such, we will ignore the stage 2 analyzer overhead in the following evaluations.

3.3.7.7.5 Cost Benefit Analysis

The overall objectives of having evaluator services is twofold. First and most importantly, it is about finding *feasible* runtime environments in which applications can provide **at least 1** session slot, with the narrow interpretation of a session slot as being an runtime environment that provides sufficient QoS/QoE for hosting one active session. For example, for a 3D low-latency real-time rendering service, many runtime environments will not be able to meet the minimal application requirements.

Secondly, evaluator services should provide also a cost benefit, amounting to a reduced cost for hosting a similar number of **useful** session slots. Useful here is key, as it is useless to deploy 3D rendering services in cheaper but excellent environments that are too far from the average user.

With respect to quantifying the first benefit, one can compare with the scenario where no evaluator service would be used. Then in worst case, actual service instances may be deployed on runtime environments without reporting any session slots and instead immediately terminating itself. As such, in that respect, the main benefit may not be in terms of runtime overhead, but in the deployment delay for deploying sufficient useful session slots in a particular region, providing a clean coordinated way instead of a trial-and-error approach.

The second benefit is by definition more easily quantifiable. When only considering this benefit, to reach the break-even point for deploying evaluator services, this cost benefit must outweigh the cost/runtime overhead for deploying the stage 1 and stage 2 evaluators. This will be evaluated in more detail in the next section.

3.3.7.8 Evaluation

In this section, we evaluate and explore how aggressively evaluator services can be deployed in order to be break-even, apart from the other benefits w.r.t. easily and cleanly finding feasible runtime environments. As concluded from the previous section, we will focus on the runtime overhead of stage 1 evaluator services, assuming $\rho_1 \approx 1$, and $C_2 \approx 0$. In case of flat rates for running stage 1 probes or fractional prices, the conclusions can be adjusted accordingly by adding the appropriate correction factor.

In Table 2, approximate values for E_1 are displayed for running an evaluator probe for a particular amount of time (i.e., 1 minute, 15 minutes and 1 hour) at a particular deployment frequency (i.e., hourly, ..., 3-yearly). For example, daily deploying a single probe and running it for about 15 minutes results in a relative probe execution time E_1 of about $1/100$, meaning the aggregate of probes are active about 1% of the time.

	frequency	1 minute	15 minutes	1 hour
extreme repetition	hourly	60	4	1
high repetition	daily	~1/1,500	~1/100	1/24
medium repetition	weekly	~1/10,000	~1/700	~1/170
low repetition	monthly	~1/43,000	~1/3,000	~1/700
one-shot	yearly	~1/525,000	~1/35,000	~1/9,000
one-shot	3-yearly	~1/1,600,000	~1/100,000	~1/27,000

Table 2 – Values for E_1 as a function of probe frequency and average probe runtime duration.

Both the repetition frequency as well as the duration of a single probe session are very application dependent. In many cases however, a probe will a one-shot deal, quickly assessing the feasibility and perhaps the stability of a particular runtime environment. A repetitive probe will only make sense in specific cases, such as (i) for doing active monitoring using statistical sampling in case of very popular and/or critical services, as well as (ii) in case of a service class probe that can be leveraged and customized for a wider range of service types.

Note also that in a distributed heterogeneous cloud, new runtime environments will frequently come online and old ones will be decommissioned. As such, even one-shot probes may have to be deployed regularly to assess these new environments.

Using all this information, we can now provide some intuition in the order of magnitude of probes one could deploy to be break-even, given some target runtime overhead R_1, probe runtime fraction E_1, probe sharing factor S_1, and average number of deployed service instances I_1. Table 3 depicts some numbers for a fixed probe runtime fraction E_1 of 1/10,000, which roughly amounts to running a probe for the equivalent of about 1 hour per year, 15 minutes per quarter, or 1 minute per week. Assuming an allowed runtime overhead of 1% and a sharing factor of 1 (meaning the same probe runtime only is used by 1 service type, then the maximum number of probes one could deploy is 100, multiplied by the number of active instances of a single service type. So in case there are on average 10 instances constantly deployed, then about 1,000 probes can be deployed across the various (globally unique) runtime environments. Combined with the pre-filtering optimizations and identification of globally identical and unique runtime environments, this means that a huge amount (if not all) of environments can be probed (once) with low runtime/cost overhead. Even for the long tail of unpopular services, running one-shot probes can still be beneficial with relatively low overhead. Note that for larger probe sharing factors or allowed runtime overheads, the number of runtime environments that can be probed increases even further.

	S_1=1	S_1=10	S_1=100
R_1=100%	N_1=10k × I_A	100k × I_A	1000k × I_A
R_1=10%	1k × I_A	10k × I_A	100k × I_A
R_1=1%	100 × I_A	1k × I_A	10k × I_A

Table 3 – Values for N_1 as a function of the maximum allowed runtime overhead R_1, a probe sharing factor S_1 and service instance cardinality I_1, for a probe runtime fraction E_1 of 1/10,000 (~1h/year, 15m/quarter or 1m/week).

When the probe runtime fraction increases, the number of probable runtime environments decreases accordingly when everything else remains fixed. Table 4 shows the results for $E_1 \approx 1/1{,}000$, which corresponds to running a probe for about 1 hour per month, 15 minutes a week, or 1 minute every day. For even higher frequencies (e.g. for doing statistical profiling and/or active monitoring), these numbers for N_1 drop even further. In these cases, only probes that are highly reusable or services that are very popular can cost-efficiently run these types of probes.

	S_1=1	S_1=10	S_1=100
R_1=100%	N_1=1k × I_A	10k × I_A	100k × I_A
R_1=10%	100 × I_A	1k × I_A	10k × I_A
R_1=1%	10 × I_A	100 × I_A	1k × I_A

Table 4 – Values for N_1 as a function of the maximum allowed runtime overhead R_1, a probe sharing factor S_1 and service instance cardinality I_1, for a probe runtime fraction E_1 of 1/1,000 (~10h/year, 1h/month, 15m/week or 1m/day).

Remember also that if $\rho_1 \approx 10$, then $C_1 = 10 \cdot R_1$, so N_1 decreases by a factor of 10. On the other hand, in case $\rho_1 \approx 0.1$ (e.g., in case of a particular flat rate), then N_1 increases by a factor of 10.

3.3.7.9 Optimization Strategy

Combining all insights and results from the previous sections, we now present a high-level optimization strategy to minimize the overhead of these probes while maximizing the useful coverage of deploying probes across runtime environments:

- First, decide on whether you need active stage 1 probes and/or stage 2 analyzers or whether a static manifest is sufficient. This effectively tries to minimize N_i to 0.
- Second, decide on whether you can leverage (the results of) an existing (repetitive) probe or not. This maximizes S_i. For stage 2 analyzers, this typically should be relatively easy to do.
- Third, filter and/or prune the various (geo)locations and/or runtime environments in which to deploy active probes.
  - Describe and/or learn the bounding boxes of the optimal local and/or global environment (i.e., smart & explicit filtering) of the target application service. For example, describe reasonable minimum and/or maximum values for memory, CPU, networking bandwidth & latency, particular accelerators (GPU, transcoder, etc.), absolute and/or relative geographical location(s), etc. This will minimize $N'' \approx \max(|V_u''|, |Z'|)$.
  - Describe and/or learn the expected optimal unit price point for the target application service. This will minimize $|V_u''|$ by optimizing ρ.
- Fourth, determine the fraction of time E_i the probe really needs to be deployed to obtain accurate enough results. This will depend on the requirements and characteristics of the application service.

- Estimate the target number of parallel instances I_A within e.g., the 3 months or year to determine whether it makes sense to invest in deploying many probes.
- Determine or set the target runtime/cost overhead (e.g., 1%) and expected cost benefit.
- Calculate whether the resulting number N_i covers all useful unique runtime environments (or in worst case a statistically relevant portion). If it does and with a large margin, you could try to relax the most stringent requirement (e.g., cost, probe runtime portion, etc.) to further improve the coverage. If it doesn't you could try to make the filtering more aggressively (for example using statistical sampling of environments), increase the allowed overhead, etc.

This (possibly iterative) set of steps will optimize the breadth and depth of evaluator services specifically tailored to the target application service and runtime environments. These steps should be (automatically) re-evaluated whenever some key parameter changes (e.g., the service popularity I_A).

3.3.7.10 Conclusions

In this section, we elaborated in more detail on the FUSION evaluator services concept, presented the basic model and extended this model by splitting the evaluator in a stage 1 probe and stage 2 analyzer service. This split facilitates reuse on the one hand, and can significantly reduce the runtime and cost overhead on the other hand, as these probes typically will only be active for a short amount of time; the cost-benefit analysis on the other hand can be done more centrally, and thus more efficiently. This mechanism also facilitates the use of multiple types of probes to be automatically aggregated into one or analyzers. We mathematically modelled the cost/runtime overhead and benefit and used this model to assess in what cases a particular deployment strategy of evaluator probes can be leveraged. Finally, this was crystallized in the formulation of an optimization strategy for service placement.

3.4 Zero-Overhead Inter-Service Communication

This section discusses the impact of minimizing the inter-service communication overhead. We evaluate and compare various mechanisms in the context of raw video streaming in between micro-services.

3.4.1 Motivation

As already discussed in Section 3.2, it may be beneficial to decompose a service into several micro-services in a heterogeneous cloud environment to be able to better match the service requirements with the available hardware and software capabilities. However, for real-time media processing applications, this means breaking down the real-time media processing pipeline into several subcomponents that are packaged and deployed separately as micro-services, for example as Docker containers, Unikernels or plain VMs. As the communication exchange in between these subcomponents of a media processing application typically includes raw media data (e.g., raw video or audio frames), serializing and streaming this data across a socket interface typically involves a significant overhead in processing requirements (e.g. encoding and decoding) or networking requirements (e.g., bandwidth and latency). For example, using efficient real-time media codecs such as H.264, one could easily reduce the network bandwidth and latency requirements by several orders of magnitude at the expense of significant computational overhead and latency. Streaming raw data on the other hand lowers the computational overhead at the expense of much higher network requirements.

As such, to avoid losing the benefits of a micro-services model for real-time media applications in a heterogeneous cloud environment, we ideally need zero-overhead inter-service communication mechanisms that however do not break down the underlying principles of a micro-service deployment model. In Deliverable D3.2, we already studied a number of inter-service communication channel optimizations in case of general inter-VM and inter-Docker communication. In this Section, we will evaluate and compare the overhead and effectiveness of various existing and custom-made inter-service communication mechanisms in the context of real-time media applications, using our Vampire prototyping framework [Va09] as real-time media processing framework, and Docker containers as packaging and deployment model.

3.4.2 Existing Technologies

In Deliverable D3.2 Section 4.5, we already studied various networking technologies for doing inter-VM communication:

- **SR-IOV based inter-VM communication**

  Leveraging the hardware virtualization capabilities of physical NIC cards for boosting networking capabilities. Virtual hardware contexts of the physical NIC are directly mapped into the VMs, allowing to bypass the host NIC drivers and corresponding overhead.

- **IVSHMEM based inter-VM communication**

  Shared-memory based inter-VM communication mechanism, modelled as PCI devices within each VM. Physical memory pages can be directly shared across VMs, and a doorbell mechanism is used for signalling between VMs. This mechanism is only available for VMs running on the same physical host and requires special communication libraries (e.g., DPDK or custom).

- **DCTCP**

  Enhanced TCP congestion control mechanism for improving throughput stability typically within data centres.

- **LibChan**

  Go-alike message passing channel for Docker containers, supporting various transports, including in-memory channels, UNIX sockets, RAW TCP, HTTP2, WebSockets, etc.

- **RoCE**

  RDMA-based hardware accelerator for low-overhead inter-host communication where the entire network stack is offloaded to the NICs and DMA is used for efficient memory block copies in between different hosts, resulting in zero CPU overhead, high bandwidth and low latency.

Another inter-VM communication accelerator that was not discussed in D3.2 is Intel DPDK, which can also significantly improve the networking capabilities (i.e., boosting bandwidth while reducing latency) in between VMs running on the same host or different physical hosts, and leverages some of the mechanisms summarized above.

In this Section, we will mainly focus on evaluating different inter-container communication technologies for container deployed on the same (physical or virtual) host. Specifically, we will compare the following network technologies across Docker containers:

- **Standard TCP Sockets**

  Our baseline inter-container communication protocol is the standard TCP networking stack for streaming raw video frames in between containers. This technology works for both intra-host as well as inter-host container communication (physical or virtual), though we will only evaluate intra-host communication in this case study.

- **ZeroMQ**

  ZeroMQ is a distributed messaging framework for efficiently and reliably exchanging messages in between applications or application components. It supports various optimized protocols for intra-process (in-memory), intra-host (UNIX domain sockets) and inter-host unicast and multicast communication. In our case study, we leveraged the intra-host communication protocol (i.e., ipc protocol) in between different containers running on the same host. Applications need to be modified to be able to leverage their APIs.

- **Speedus**

  Speedus is an acceleration library for the standard Socket API that can be preloaded along with your application, typically providing higher throughput and lower latencies for applications or containers running within the same networking namespace or host. They also provide a PRO version that tries to use shared memory to achieve even higher throughput or lower latencies. A key advantage is that the application does not need to be modified; the acceleration library only needs to be preloaded when starting the application(s).

- **POSIX SHM**

  Within the same host, containers can also communicate with each other via shared memory if properly configured. We implemented and evaluated a custom SHM-based inter-service communication protocol within our Vampire Media Processing Framework (which was also used for some of the FUSION application service prototypes such as the EPG service) to evaluate and compare the performance with the other approaches. We implemented different mechanisms, involving a different amount of copies that need to be done for exchanging data. This approach only works within the same host and requires a rewrite of the application(s) to leverage the dedicated protocol.

3.4.3 Case-Study: Raw Video Streaming

In this section, we evaluate different inter-container communication protocols using raw video streaming as a case study. In this setup, we investigate the impact of streaming raw video in between two Docker container services across various communication protocols and implementation options. The application setup is depicted in Figure 54:

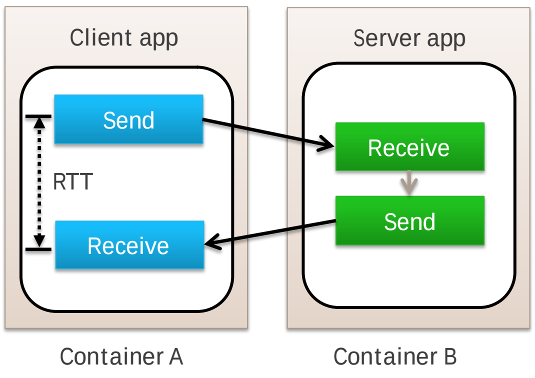


Figure 54 – Application Setup: a pair of containers containing a Client and Server test application.

The test setup comprises a ping-pong test consisting of two applications, each running in their own Docker container on the same host. A client will continuously send raw video frames over a selected communication channel. A server receives a raw video frame and sends the frame back to the client. The client measures how long it takes to send and receive the raw video frame and also measures the total throughput (in FPS).

Note that in typical media processing pipelines as shown in Figure 55, video frames are only forwarded to the next component and are never sent back to the original component. As such, all throughput results in this section can be doubled and latency results can be halved to know the effective throughput and latency for transmitting raw frames in between two services running in separate containers.

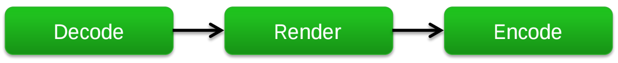


Figure 55 – Simple media processing pipeline (e.g., EPG service, transcoding service, etc.).

3.4.4 POSIX SHM Copy Modes

This section discusses the mechanisms and implementation details we used for implementing a custom POSIX shared memory based communication channel in our Vampire Media Processing Framework.

3.4.4.1 Mechanism

We implemented three main mechanisms for exchanging data (i.e., raw video frames) in between the client and server application, involving a decreasing amount of explicit buffer copies in between the applications and the shared memory regions, but with an increasing amount of required applications-specific support. The three main modes are depicted in Figure 56.

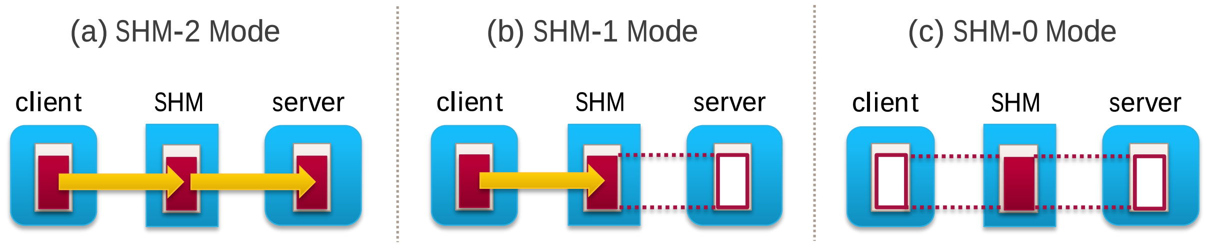


Figure 56 – Different SHM Copy modes: (a) 2-copy mode, (b) 1-copy mode, and (c) zero-copy mode.

In the SHM-2 mode, one or more dedicated shared memory buffers are directly mapped into the client and server applications. The client copies the raw frame data (possibly packetized) into the shared memory buffer, and the server copies back the frame date from the shared memory buffer into a local memory buffer after having received a signal that new data is available (e.g., via a shared semaphore or other doorbell mechanism).

In this mode, these shared buffers can be of any size (i.e., smaller or larger than the application frame data) and are used as intermediate buffer to copy data between client and server. This independence of buffers allows for the client and server applications to remain relatively agnostic and require at best only minor changes to be able to use such SHM-based communication channel. However, in this mechanism, all data is copied twice (cfr. the two yellow arrows in Figure 56(a)), resulting in non-negligible memory bandwidth overhead and CPU overhead as well as possible trashing of the caches in case of high-resolution and/or high frame rate raw video frame transmission. Note that this mechanism is for example also used by Speedus EP as backend for exchanging data between applications over the standard socket interface.

In the SHM-1 mode, the client still copies the raw frame data explicitly into the shared memory buffer. The server on the other hand, rather than copying the shared data into a local buffer, it wraps the shared data frame buffer into a local data structure and directly refers to the shared data frame buffer. This way, a second explicit copy is avoided and the server pipeline will directly read from the shared data frame buffer when processing the received video frame.

This avoids that an explicit second memory copy needs to be done. However, this does have some consequences on size and use of the shared memory buffer, as well as the serialization. First, as the shared data frame buffer is not immediately copied but could be in use longer in the server pipeline application, each shared memory frame buffer will typically be longer in use. Moreover, appropriate (shared) memory management and pooling mechanisms need to be implemented so that a shared memory buffer is only reused by the client when it is not in use anymore by the server pipeline. This means that typically more shared memory buffers will have to be used in parallel. Also, the shared memory buffers typically need to be large enough to contain an entire raw video frame so that the entire frame is directly accessible from within the server pipeline. Consequently, in this mode, we are trading off memory bandwidth and CPU consumption overhead with memory capacity requirements and application complexity and awareness.

The SHM-0 mode finally goes even one step further than the SHM-1 mode. In this mode, the client application allocates all its shareable frame data (i.e., non-local video frames that will be transmitted to another application) directly within some shared memory space, and wraps these buffers into its local data structures. The media pipeline components in the client application render directly into these shared memory data structures. To transmit a rendered frame, the client only signals the server about the existence and location of a new available frame. Note that this signalling could be done in another shared memory space, referring to the memory space where the original frame is stored. Then, like in the SHM-1 approach, the server simply maps the client shared memory frame into its own local data structures. The huge advantage of this approach is that no frame data needs to be copied at all, making this mechanism also independent of the resolution of the video frames. This approach however typically requires even more shared memory capacity and a smart cross-application memory pooling mechanism so that the client application only reuses some shared memory buffer when the server does not need it anymore.

Although modes SHM-0 and SHM-1 introduce additional complexity w.r.t. local and shared memory buffer management, as will be illustrated in the next Section, this can be done without impacting the rendering pipeline components.

3.4.4.2 Implementation

Although SHM-2 can be implemented with limited to almost no modifications to the applications (e.g., Speedus-EP implementation), the other two approaches do require an increasing amount of application support. For one, the frame buffer data layout need to match between client and server. This means both client and server need to either use a standardized format or must be co-developed: if the client uses a packed BGRA32 format but the server uses a planar RGB format, then at least one conversion needs to be done, nullifying any benefits of these copy-saving approaches.

Secondly, careful buffer management techniques must be used in the server and possibly client application to allow for smart shared memory pooling mechanisms. If not done in a smart way, this could result in very specific inter-service communication code to be spread across all pipeline components of both client and server application, thereby significantly increasing the overall complexity of the code as well as making it strongly linked to the supported inter-service communication channel. For example, if client and server happen to run on different hosts or on a host that does not support shared memory communication channels, then we still want the applications to be able to communicate with each other with some other communication channel (e.g., standard sockets).

As such, when implementing the SHM-1 and SHM-0 communication channels in our Vampire framework, we exploited two key enabling features of the framework:

- To enable SHM-1 support, we exploited the advanced (image) memory pooling capabilities provided by the framework. In our framework, our key *t_image* data structure for representing images already supported a memory pool mechanism with reference counting not to constantly have to (de)allocate or worry about creating and managing these image structures and their frame buffers.

  As such, in the server application, the only thing we had to add, was a special SHM-aware memory pool that still creates local *t_image* data structures, but references *some* SHM image for its frame data, including proper reference counting and SHM signalling (also at the client side in the transmission component), but without requiring any knowledge or changes to any other component in the client or server pipeline components using *t_image* objects; they remain agnostic about the source or destiny of its underlying frame data.

- To enable SHM-0 support, we exploited the advanced accelerator backend capabilities of the Vampire framework. This accelerator backend allows for system-aware accelerated implementations of particular key functions. One typical example is a YUV420 to RGB conversion, where, depending on the actual runtime environment, it may be preferable to use an SSE2, SSE4.2, AVX2, Neon or OpenCL accelerated implementation of the same function.

  In this example, we leveraged the backend to provide a SHM-based default imagepool creation function, allowing to but automatically allocate image frame data from some SHM buffer, possibly using a custom SHM memory map as well as possibly only applicable for selected pipeline components (e.g., only those producing non-local image frames).

  As such, all this complexity of buffer management, combined with the smart pooling capabilities, is completely handled and managed behind the scenes w.r.t. all pipeline rendering components and does not require any changes to any of these existing/legacy components.

Finally, to enable backward compatibility and fallback support to standard Socket-based inter-service communication, we implemented an auto-discovery protocol over standard TCP sockets, where a client announces its availability of SHM-support, and SHM-aware and enabled servers signal back to the client to switch to SHM-based communication of possible. In these scenarios, if the client, server or cloud subsystem do not support SHM (or a compatible image protocol), then they simply (continue to) communicate over standard sockets (possibly accelerated via e.g. Speedus).

3.4.5 Methodology

All experiments were done on a dual-socket Xeon E5-2690v2 HP DL380 G8 Server, using an Ubuntu 14.04.3 Server OS. Both the client and server test applications were wrapped into Docker containers, using Docker 1.8.1. The video stream was serialized as a raw Y4M video stream in the BGRA32 data format. We ran the experiments at different video resolutions (360p, 720p, 1080p, etc.). We experimented with different loads by increasing the number of parallel ping-pong test applications running on the same host, to measure the impact of cross-interference (both in the network stack as well as in memory). We experimented with different host deployment optimizations, ranging from on-demand CPU throttling without NUMA pinning towards performance-optimized CPU throttling with NUMA pinning enabled.

With respect to the different communication channels, we evaluated the following strategies:

- TCP: standard TCP sockets
- Speedus-Lite: the default ‘lite’ version of Speedus, not including SHM-based acceleration
- Speedus-EP: an evaluation copy of the PRO Extreme Performance ‘EP’ version was used here
- ZMQ-2: an optimized implementation of ZMQ, involving a 2-copy mechanism. Although the implementation itself is 0-copy w.r.t. the application-ZMQ interface (i.e., using *zmq_msg_init_data* for sending application buffers without copying, and using the received zmq-buffer-data directly inside the raw image frame data structures using the same smart memory pooling mechanism as used for the SHM-1 approach in Section 3.4.4.2), all data still needs to be serialized over a UNIX domain socket, involving a 2-copy mechanism (i.e., write-to-UNIX-socket, and read-from-UNIX-socket), and as such is not truly zero-copy. We also evaluated a ZMQ-3 version, not involving this smart memory pooling mechanism, but this obviously performed even worse and is not included in these results.
- SHM-2: The basic 2-copy SHM-based custom Vampire implementation, as described in Section 3.4.4.2.

- SHM-1: The 1-copy SHM-based custom Vampire implementation, as described in 3.4.4.2, using the smart memory pooling mechanism for receiving frame data.
- SHM-0: The true 0-copy SHM-based custom Vampire implementation, as described in 3.4.4.2, using the SHM-1 smart memory pooling mechanism for receiving frame data combined with smart memory pooling for allocating rendering frames and exchanging these SHM-based rendering frames on top of the SHM-1 approach.

For each inter-container communication strategy, we also evaluated the impact of different settings and options on the throughput, latency and CPU overhead. Key options we evaluated (though not all obviously are relevant for all strategies):

- MTU size: given the size of these frames, we tried different MTU sizes: 1500, 9000 and 64k, which are all possible on the localhost interface.
- Send/Receive buffer size: sending and receiving frame data per scanline (i.e., 1 send()-call per image scanline) or per entire frame (i.e., 1 send()-call per frame).
- # Threads: number of parallel threads in the server.
- # SHM buffers: number of SHM buffers used in parallel for exchanging data.
- Polling mode: enable or disable active polling instead of blocking mode (cfr DPDK).

3.4.6 Evaluation

This section evaluates the impact of the various inter-service communication channels between services deployed as Docker containers on the same host using the example benchmark application as described in Section 3.4.3. Unless otherwise stated, the results shown are for a 720p frame resolution running the system in high-performance mode.

First, we will present and discuss the overall results, without focussing on the specific impact of the various per-communication channel optimization options (e.g., MTU size, etc.). In the second part of this evaluation, we then zoom in on the specific impact of these settings on the overall throughput and overhead.

3.4.6.1 Overall results

In Figure 57, the overall throughput and roundtrip latency results are depicted for the various inter-service communication (ISC) strategies when running a single pair of client-server test applications directly on the bare host (i.e., not wrapped in Docker containers) using localhost as network interface. Note that in this and the following graphs, the various data points per ISC strategy corresponds to the various configuration options (e.g., MTU size, packet size, etc.), which are studied in more detail in the following section. The results for SHM-0 are also omitted from the following graphs and are discussed at the end of this section, as they do not easily fit in these graphs.

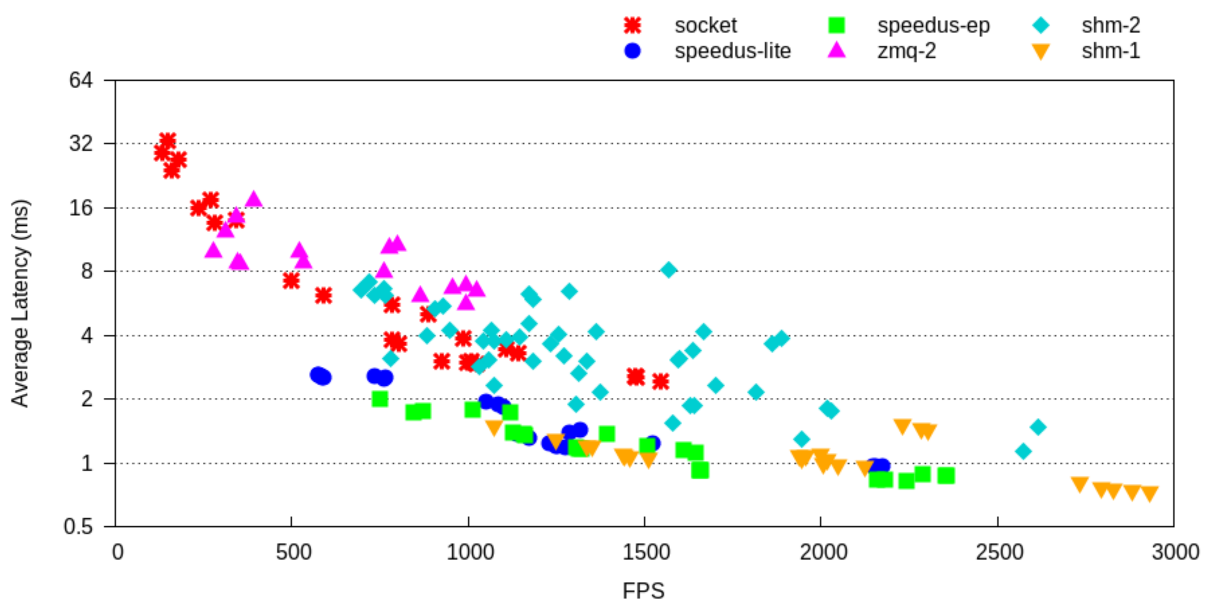


Figure 57 – Overall throughput and average roundtrip latency for the various inter-communication strategies when running a single client-server pair on the bare host via localhost.

In this graph, a number of things can be observed. First, there is a huge difference in frame rate and roundtrip latency between the worst (i.e., default, as we will show in the following section) socket-based communication strategy and the best SHM-1 optimization strategy: throughput can be increased by about a factor 20 and roundtrip latency can be even reduced by a factor 40, from about 32 ms to only about 0.8 ms.

A second key observation is that for each ISC strategy, carefully optimizing the various options has a huge impact on the overall maximum frame rate, and (especially for the socket-based strategy) also a significant impact on the average roundtrip latency. Optimizing the socket-based ISC strategy can improve the throughput by a factor 10 (achieving half the performance of SHM-1) and can improve the latency by a factor 15. Note that absolute roundtrip latency values are extremely important w.r.t. low-latency real-time media services: introducing several (tens of) milliseconds of additional (unnecessary) latency in between service components contradicts the overall objective of FUSION w.r.t. low-latency interactive real-time cloud services.

A third key observation is that ZMQ-2 seems to perform quite badly, and is even surpassed by the most optimal socket-based implementations, whereas Speedus-Lite and Speedus-EP on the other hand seem to perform extremely well, with low roundtrip latencies across the board and with very high throughputs in both cases. Notice that running bare metal using localhost, Speedus-Lite can perform almost as good as Speedus-EP, which is using shared memory as communication channel. Also, reducing the number of copies to only one (i.e., SHM-1) still has an advantage w.r.t. absolute throughput, but not by a factor 2.

In Figure 58, the same results are shown, but now in case the client and server application both are deployed in a separate Docker container instance, but still use the same networking namespace so that they can still communicate with each other over localhost, through they may still be isolated from other container applications w.r.t. also their networking stack. As such, this graph shows the baseline impact of running the applications in containers, but excluding the impact of having separate network interfaces. As can be observed, the results are very similar to running the applications bare metal; as such the overhead of containerization is minimal w.r.t. these ISC strategies.

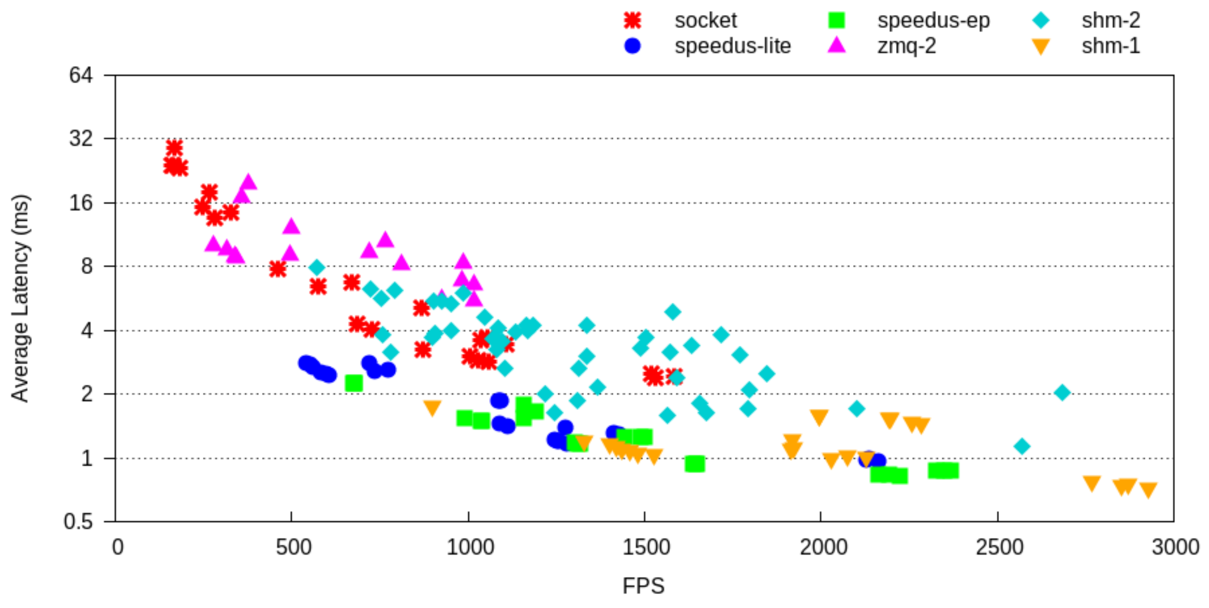


Figure 58 – Overall throughput and average roundtrip latency for the various ISC strategies when running a single client-server pair in their own Docker container but communicate via localhost.

Figure 59 then depicts the impact of running each container in their own networking namespace, connected to each other via the standard Docker Linux bridge. As can be observed, this mainly impacts the socket and Speedus-Lite ISC strategies. As the Speedus-Lite approach only give a performance advantage in case both services communicate over some localhost interface, in this setup, there is no performance advantage anymore compared to the standard socket strategy. The socket strategy however is impacted by the additional networking overhead caused by the separate networking namespaces, the *veth* pairs and the Linux bridge in between them: the baseline performance deteriorates even further to below 100 FPS and the worst-case roundtrip latency increases to about 50 ms. In the optimal socket case, throughput and latency are also slightly impacted, but still perform reasonably well. Note that, as the other strategies are independent from the local networking interface, their performance remains roughly the same.

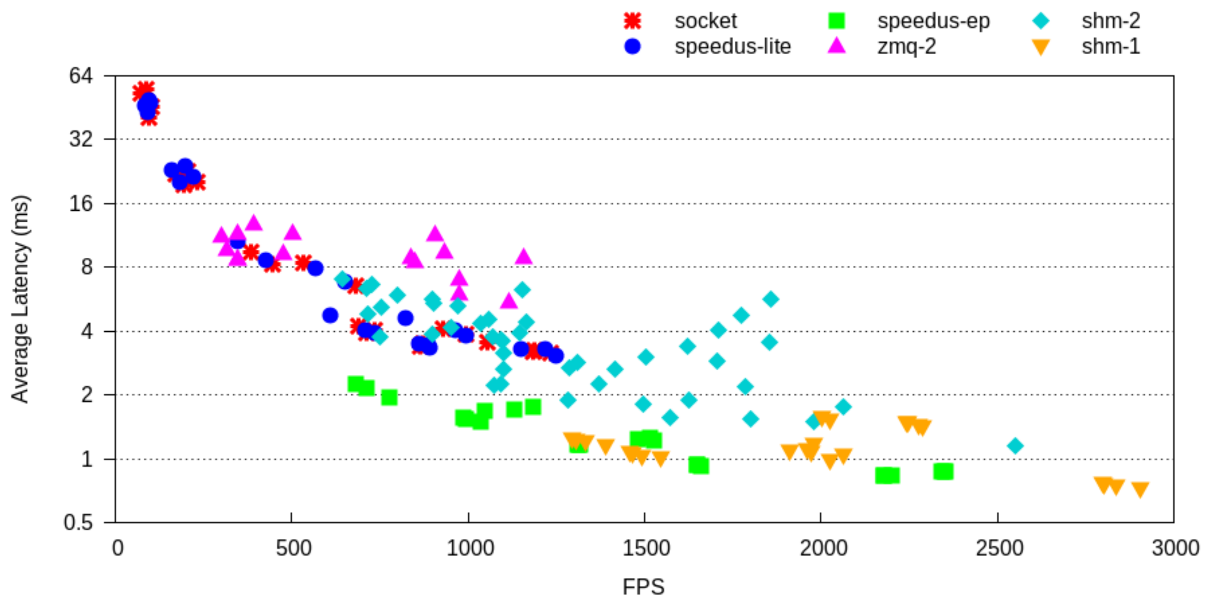


Figure 59 – Overall throughput and average roundtrip latency for the various ISC strategies when running a single client-server pair in containers and communicate via the standard Docker bridge.

In the previous results, we looked at the maximum throughput each of the ISC strategies could achieve, irrespective of the corresponding CPU overhead it introduces. In Figure 60, we now show the CPU-normalized throughput that each strategy can achieve. In other words, how much frames per second can one achieve per fully-utilized CPU core. When looking at these results, one can observe than SHM-1 is more than two times as CPU-efficient than the other approaches, meaning the CPU overhead is less than half of the overhead introduced by the other strategies. This is the impact of only having to actively copy each frame once per transmission instead of at least twice.

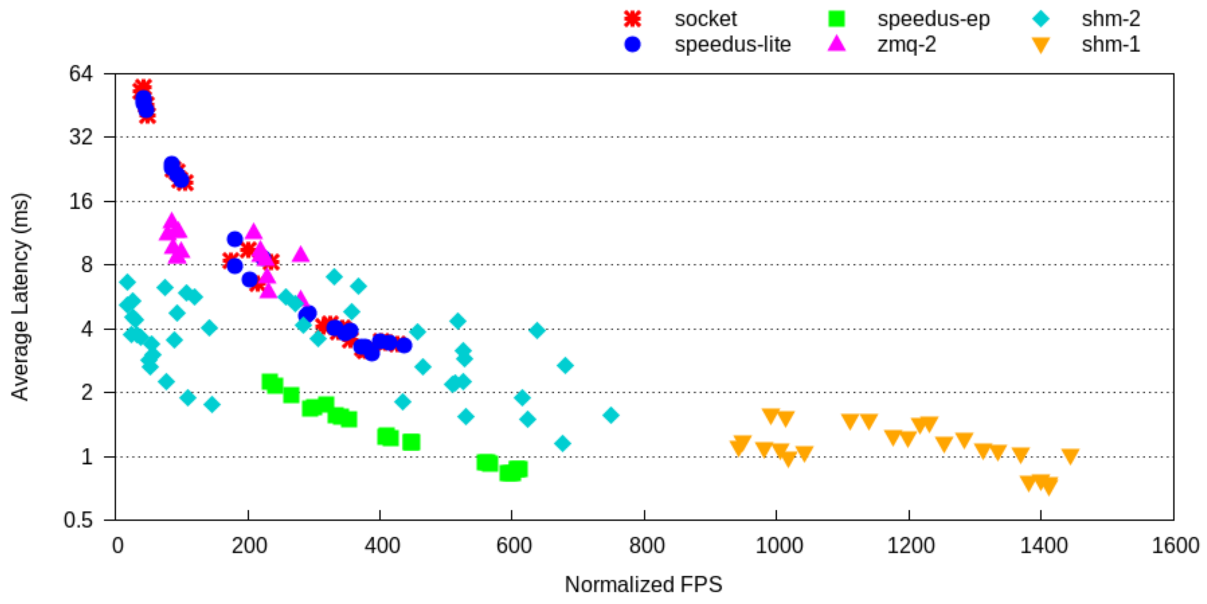


Figure 60 – CPU-normalized throughput and average roundtrip latency for the various ISC strategies when running a single client-server pair in containers using the Docker bridge.

Consequently, although the absolute performance of SHM-1 compared to e.g. Speedus-EP is not that high for a single instance pair, the performance efficiency is twice as good. This can also be observed when increasing the overall load on the system, by increasing the number of independent parallel client-server pairs on the same physical system. This is shown in Figure 61, where we demonstrate the impact of running 4, 10 or 20 client-server pairs in parallel on the same physical host on the CPU-normalized throughput and roundtrip latencies. Two key observations can be made here. First is that the benefit of SHM-1 significantly increases as the overall CPU and memory bandwidth of the system are getting saturated. A second observation is that, as the system is under higher load, the roundtrip times start increasing as well, due to CPU and memory bandwidth saturation. However, the relative differences in latency remain very roughly the same.

Obviously, these tests only stress the inter-service communication aspect of a real-time media processing application. In typical scenarios, the actual required frame rate will be much lower (25-100 FPS). In these cases SHM-1 and Speedus-EP have the lowest corresponding roundtrip latencies as well as CPU overheads. Although SHM-1 has a clear advantage w.r.t. the overall CPU overhead, the fact that for Speedus-EP, the application itself did not have to be modified, is also a clear advantage. It is up to the application developer to decide whether it makes sense to invest in implementing a SHM-1 (or even better a SHM-0, see below) ISC communication strategy within its application framework.

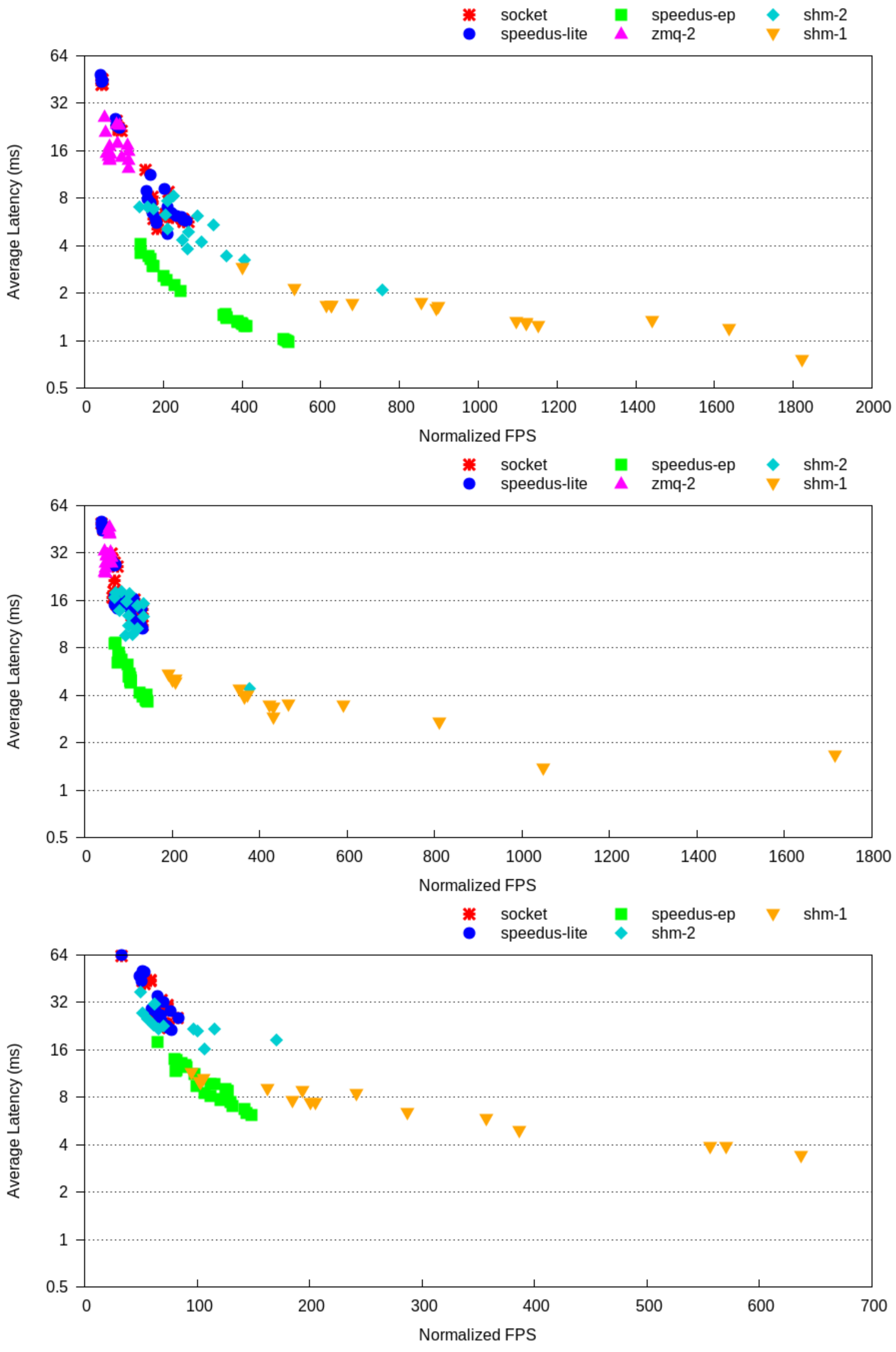


Figure 61 – CPU-normalized throughput and average roundtrip latency for the various ISC strategies when running a 4, 10, and 20 client-server pairs in containers using the Docker bridge, respectively.

So far, we explicitly left out the results for SHM-0, as they are several orders of magnitude better than any of the other approaches due to the fact that this approach does not involve any copying operation and boils down to merely a signalling operation. Note also that, as mentioned earlier, SHM-0 is independent from the frame resolution, meaning communication overhead, latency and throughput remains the same as the resolution increases (4k+ video).

We implemented two variants for the SHM-0 signalling, both using a shared semaphore as signalling mechanism. Other mechanisms could also be used to further improve throughput and reduce latency. One is a blocking mode where the Receive pipeline component blocks until a new frame is available (*sema_down*); the other is a polling mode (cfr also DPDK) where the Receive component continuously polls whether a new frame is available (*sema_poll*). Note that at nominal frame rates, a blocking mode typically makes most sense.

In the blocking mode, the maximum throughput using a single client-server pair is about 230,000 FPS with a roundtrip latency of 10 μs. In the polling mode, the maximum frame rate increases up to 1.3 million FPS with a roundtrip of merely 2 μs.

Clearly, this strategy largely outweighs all other strategies discussed before. However, this approach also puts quite strong requirements on how the application manages its internal and shared buffers. Although this can be implemented in a clever way as discussed in Section 3.4.4.2, this mechanism is obviously much less flexible compared to e.g. Speedus (EP). For dedicated real-time media applications written in specialized frameworks such as the Vampire Framework however, the SHM-0 strategy allows to implement a fine-grained micro-service model where many components are deployed in their own containers on their own environments, with almost zero communication overhead.

3.4.6.2 Detailed results

In this section, we will zoom in on the impact of various configuration options and knobs for improving the performance of the various ISC strategies. In these results, we again assume a single client-server pair, running the system in high-performance mode using 720p raw video frames.

The first graph in Figure 62 depicts the impact of the MTU size, containerization and networking option when transmitting frames per scanline (also in the SHM modes) using only a single server thread and a single buffer, without active polling. This represents roughly a baseline configuration setup for all ISC strategies. Many of the observations done in the previous Section can be seen in this more detailed graph as well, such as the big performance difference between socket and the speedus and shm-based strategies, the relatively low performance of zmq and the performance impact of the Docker bridge mode on the socket-based strategies. A key new observation on the other hand is the impact of the MTU size used for the localhost interface on the performance, and the symmetry w.r.t. the localhost performance of Speedus-Lite: when using a 64k MTU size, the performance matches that of Speedus-Lite, meaning that most likely one of the key optimizations of the Speedus library w.r.t. this use case scenario is to use a similar strategy when transmitting large packets in between two applications.

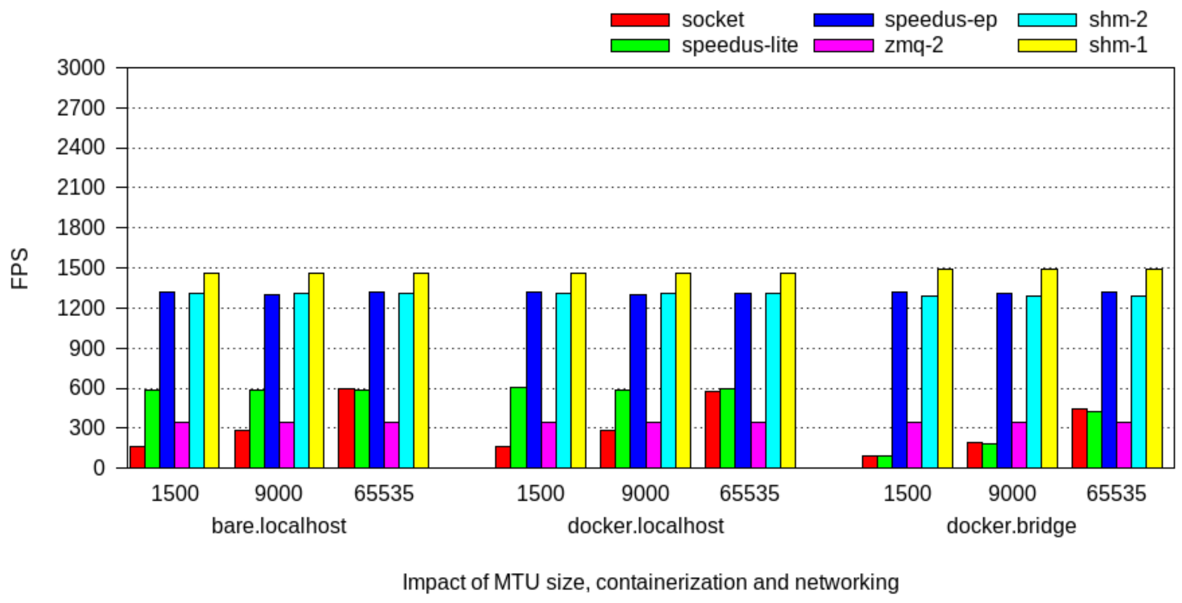


Figure 62 – Throughput results for the various ISC strategies when running a single client-server pair, using different MTU sizes, containerization and networking options, using per-scanline transmission mode, a single server thread, NUMA enabled, a single receive buffer and no polling.

Figure 63 shows the throughput results for another point in the configuration space. In this setup, we send an entire frame at once (i.e., using a single *send* syscall or using an entire SHM frame). We also increase the number of receiver threads and receiver buffers to increase throughput through better pipelining. Two key observations can be made here. Firstly, the overall throughput results are much higher than in the previous Figure due to much less overhead, especially w.r.t. the number of *send* syscalls. Secondly, the MTU size in this case has become irrelevant; in this setup, all ISC strategies perform more or less at their optimal: SHM-1 has the highest absolute throughput, closely followed by Speedus-EP and SHM-2. The default Socket-based implementation is up to about a factor 2 less performant, but obviously does not require application modifications. ZMQ performance is the worst across the board.

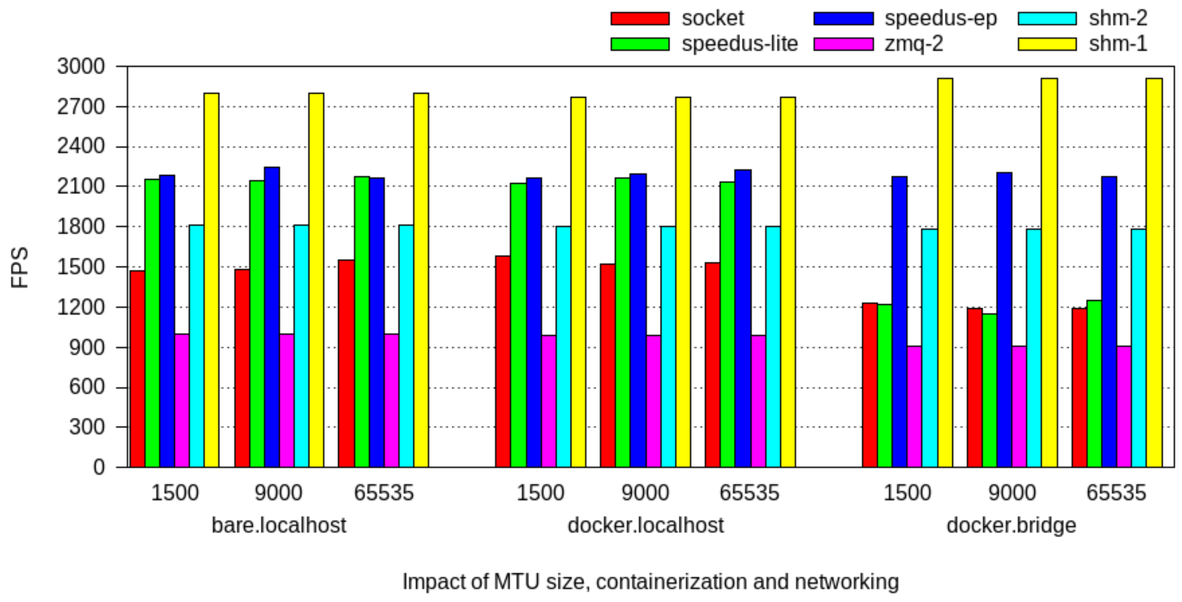


Figure 63 – Throughput results for the various ISC strategies when running a single client-server pair, using different MTU sizes, containerization and networking options, using per-frame transmission mode, a two server threads, NUMA enabled, two receive buffers and no polling.

Figure 64 then shows the CPU utilization overhead for the same configuration setup as used in the previous Figure. In this graph, 100% CPU means that one 1 CPU core is being occupied for 100%. In this setup, most strategies seem to have a similar CPU utilization overhead. SHM-1 is much more efficient, and interestingly, ZMQ requires the highest CPU utilization even though it has the worst performance.

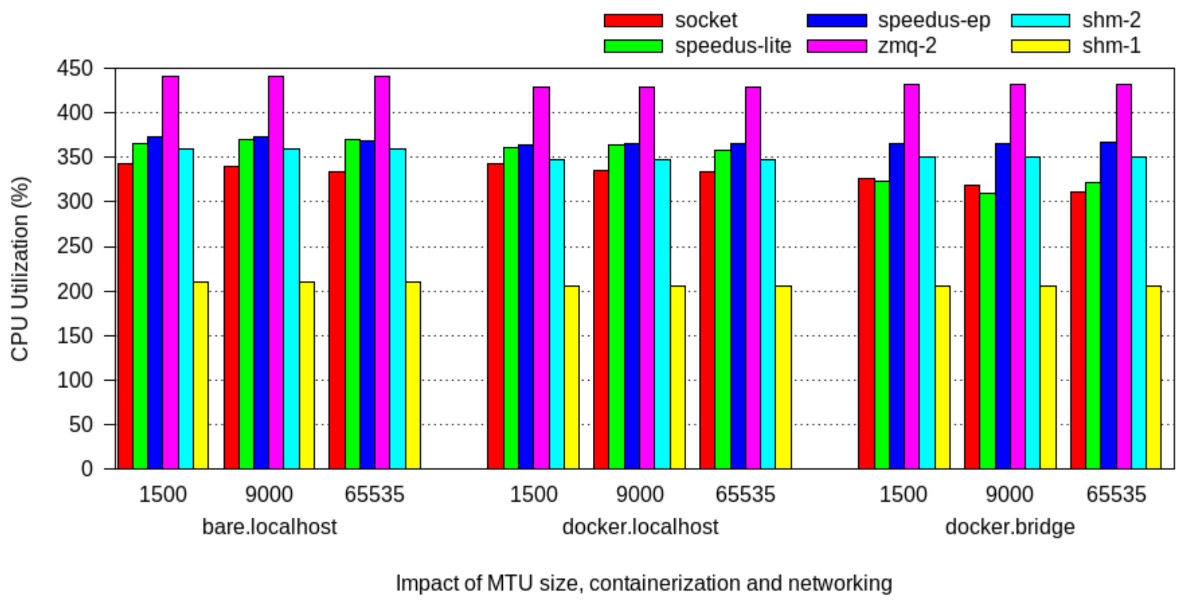
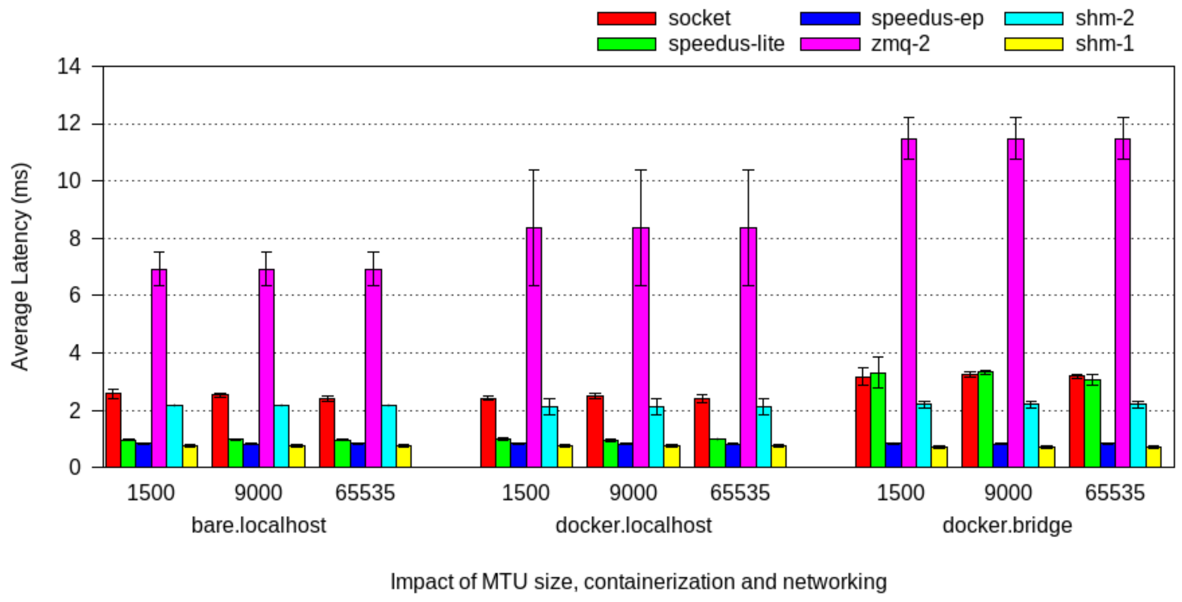


Figure 64 – CPU core utilization (%) for the various ISC strategies when running a single client-server pair, using different MTU sizes, containerization and networking options, using per-frame transmission mode, a two server threads, NUMA enabled, two receive buffers and no polling.

As a final result, we also show the average latency and standard deviation results (as error bars) for the per-frame multi-buffer configuration setup of the previous graphs in Figure 65. The upper graph shows the results for a single client-server pair, whereas the lower graph shows the results for 20 independent parallel client-server pairs. In the first graph, two clear observations are (i) the relatively high roundtrip latency and jitter for ZMQ, and (ii) that the jitter of the other strategies is very low in this isolated test case. In the second graph, the average roundtrip latencies are much higher on average due to saturation, as well as the overall jitter of the various approaches. Especially Speedus-EP seems very sensitive to jitter in this saturated setup, especially compared to the SHM-2 and SHM-1 strategies. SHM-1 clearly has the lowest jitter as well as roundtrip latency.



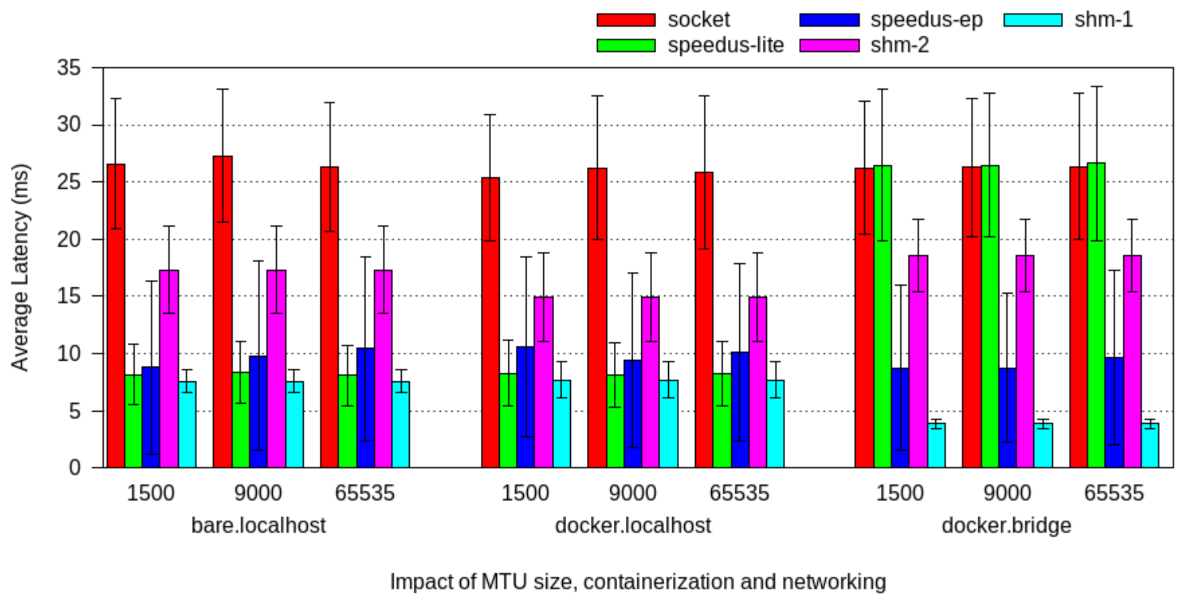


Figure 65 – Average latency and standard deviation for the various ISC strategies when running a single client-server pair, using different MTU sizes, containerization and networking options, in the pipelined per-frame configuration settings, for 1 and 20 parallel client-server pairs, respectively.

3.4.7 Conclusions

In a lightweight composite service model, efficient inter-service communication is essential. In this section, we extended the initial evaluations already started in Deliverable D3.2 and applied them on a raw videos streaming case-study. We evaluated and compared a number of existing and custom ISC mechanism in terms of performance, efficiency as well as implementation overhead/impact. Obviously, a true zero-copy mode, in which basically only a pointer needs to be passed along, is ideal, but also has most requirements w.r.t. how buffer management in the applications should be done. We did present some key design principles to minimize the impact on the existing application code based on a reference implementation in our Vampire framework. Otherwise, in terms of raw performance, Speedus EP performs exceptionally well compared to the other SHM-1 and SHM-2 approaches, and has the key advantage that the application itself remains unchanged. When normalizing the performance with the CPU overhead, the SHM-2 still has a large advantage in performance efficiency, especially when multiple service instances are active in parallel. Nevertheless, we clearly demonstrated that (extremely) efficient inter-service communication of raw data is definitely feasible, but may require additional development effort to reduce the lowest overhead.

3.5 Maximizing Service Node Density & Scalability

In this section, we present the results of a series of experiments that were performed w.r.t. feasible density numbers for deploying large amounts of virtualized services on the same physical host. In this study, we focussed more on control-plane services and investigate and compare the behaviour and impact of deploying full VMs versus lightweight containers, derive their key bottlenecks and investigate various optimizations for improving the overall density.

3.5.1 Motivation

When cloudifying physical functions (such as for example a CPE, STB or gaming console), a fundamental question is how many of these virtualized and cloudified functions one can deploy on a particular amount of physical cloud resources. For example, if in an extreme case one were only able to deploy a single cloudified instance of a CPE or STB on a classical cloud node, one would need too many physical cloud resources, negating any benefits of a cloud solution. Consequently, a fundamental question is to explore the other extreme w.r.t. how many virtualized instances one could deploy on the same physical node before some limitation or bottleneck starts popping up. Especially for control-plane functions, the more one could squeeze on the same physical resources, the lower the overall costs for running these cloudified versions, especially if one needs to mass deploy these functions.

Note however that in our FUSION architecture, we will typically not encounter such extreme densities, for two key reasons. First, the envisioned FUSION services are often real-time demanding data-plane services for which only a limited number of instances can reasonably be fit onto the same physical resources. Exceptions to this are for example game servers, which only need to synchronize the state between various game rendering clients and for which higher densities are possible. Secondly and more importantly, in FUSION, we propose the concept of session slots for easily sharing physical resources as well as application resources across multiple sessions of the same service instance. As demonstrated in [AFM16], this can significantly reduce the resource consumption overhead. At the same time, this means that only a small amount of virtualized multi-session-enabled service instances need to be deployed and managed on the same physical machine.

Nevertheless, this study below demonstrates the various issues of drastically increasing the density of VMs and lightweight containers for a relevant function. We encountered various unexpected issues and limitations that had to be overcome or circumvented to achieve high densities, especially in case of Docker containers.

3.5.2 Methodology

3.5.2.1 Application

To evaluate the density and performance behaviour of (KVM-based) VMs and Docker containers, we used a standard 12.04 Attitude Adjustment OpenWRT environment, representing a basic CPE function, and incorporating the OpenWRT rootfs with the default services enabled and running (i.e., luci, udhcp, ntp, dropbear, etc.).

This OpenWRT environment was packaged both in a tiny KVM-based VM image as well as a Docker container. The VM image size is about 50 MiB, and we allocated 32 MiB of VM memory per running VM instance. In all tests, we used the standard QEMU virtio networking drivers. The Docker container was built using the instructions described in [He16], and is about 6 MB in size. We used the default Docker networking bridge.

3.5.2.2 Evaluation Platforms

The evaluations we performed on three different evaluation platforms at two different moments in time. For the first test platform, onto which we ran the initial extreme density evaluations, we used a SuperMicro Opteron 6174 2x12 core 2.2 GHz with 64 GiB RAM, running Ubuntu 12.04.4 with the 3.11 default kernel and the system set to high performance mode. We used QEMU 1.7.0 and a custom built LXC 1.0.x version along with Docker 0.11.1 and the default AUFS (or DeviceMapper) overlay driver enabled.

For the second test platform, we used an Atom Silvermont C2750 microserver, containing an 8-core Atom server, with 32 GiB installed, running the same OS, QEMU and Docker versions.

For the third test platform onto which we ran the performance evaluations, we used a more recent Xeon E5 2690v2 2x10 core 3 GHz with 64 GiB RAM, running Ubuntu 14.04.3 with the 3.19 default kernel, and the system also set to high performance mode. We used QEMU 2.1.2 and the default Docker 1.9.1 engine, now with the OverlayFS overlay driver enabled.

3.5.3 Extreme Density Study

In this study, we simply deployed a large number of idle VM or container instances in parallel and observer the system behaviour.

3.5.3.1 KVM VM-based Density Results

In case of the VMs, the memory RAM capacity is the main bottleneck: with 64 GiB available memory and 32 MiB per VM instance, it is clear that without memory oversubscription, one can *only* run up to 2k instances on the test platform.

When deploying those 2k instances onto that platform in parallel across all 24 cores, the total boot time of all 2k instances to be up-and-running was about 4 minutes. After boot, each of the instances are accessible from another machine with good response time. Note that the net boot time of a single VM on an idle system is about 10 seconds. With respect to disk space, when using the QEMU-snapshot mode or using the QCOW2 backing file, the total disk space utilization for all 2k instances was less than 1 GB. Using the 50MiB OpenWrt baseline backing file VM, the runtime VM disks were only 500kiB in size each. Obviously, as more files would be changed over time, more disk space would be consumed. Note that the disk space requirement when not using such backing file or snapshot would increase to 100 GiB.

After having deployed those 2k VM instances, the idle load is about 2% user and 10-15% system time, meaning that the total idle KVM overhead is about 3-4 out of the 24 available physical cores. We did not try to optimize via NUMA or interrupt pinning. Note that when running 2k instances across 24 cores, this corresponds to almost 100 instances per core, meaning each VM instance on average can only have about 1% CPU core utilization, corresponding to about 200-300 MHz per VM on our initial test platform. So, although the memory is the main bottleneck in the idle scenario, the real bottleneck very quickly will be the CPU sharing when the per-instance load is increased.

In order to try to overcome the memory capacity bottleneck, we also experimented with the Kernel SamePage Merging (KSM) optimization option available in Linux and QEMU. KSM is a feature that allows to share identical physical memory pages across different processes (or VMs), reducing the total amount of physical memory required on the physical system. A Linux kernel module that can be properly configured actively scans the physical memory pages to find identical pages, and if found, collapses them onto a single read-only memory page. The module can easily be reconfigured in */sys/kernel/mm/ksm/* to tweak how aggressively the module needs to scan for such pages. Note that only memory pages allocated with the *MADV_MERGEABLE* flag will be taken into consideration, to reduce the search space, and QEMU must be enabled to allocate memory with this flag. A bit caveat of this mechanism is that the actual saved memory can vary over time, as the amount of shareable pages can change over time: pages (typically containing mutable data) that originally were identical may become unique/private over time due to one or more changed bytes.

When we apply KSM on our 2k running VM instances, about half of the physical memory can be recuperated, thus allowing for a 1:2 memory oversubscription without requiring any swap space to be used. Specifically, there were 276 MiB of shared pages, resulting in 35 GiB of saved physical memory, 8 GiB of unshared/unique memory and 800 MB of volatile memory. For those shared pages, the average sharing factor is about 128, meaning each page is shared on average 128 times.

With this optimization, we subsequently further increased the total number of VM instances. Important to note there is that in this case, we cannot start all instances too rapidly at the same time, as the KSM daemon needs some time to find and merge shareable pages. With 3k instances deployed, the CPU overhead was about 2% user time and 18% system time, with 100% CPU core utilization for the KSM daemon. The memory sharing was 370 MB shared pages, resulting in 50 GiB saved memory, 12 GiB unshared memory and 1.4 GiB volatile memory. For 4k instances the CPU overhead was about 2% user time and 23% system time (with the same KSM daemon overhead settings). The memory sharing in this case was about 460MiB in shared pages, resulting in 62 GiB saved memory, 15 GiB unshared pages and about 1.8 GiB volatile memory.

On the Atom Microserver, we also managed to start up to 2k VM instances, using the KSM optimization, saving about 26 GiB of physical memory. The overall booting time was about 20 minutes

, which is explained by the availability of only 8 light-weight energy efficient CPU cores, of which 1 was fully occupied by the KSM daemon. The idle load about 5% user time and 25-30% system time.

3.5.3.2 Docker Container-based Density Results

In case of the OpenWRT container image, we perceived a bug in the AUFS kernel overlay driver, resulting in only a decreasing number of instances that can be deployed before errors occur. At best, we were only able to deploy about 20 instances of our OpenWRT container images using AUFS, which is very disappointing. In the old setup using the old Docker 0.11 daemon, we subsequently switched to DeviceMapper as overlay file system. There, we hit a devicemapper limitation of 256 instances of the same image. With the container images being 6 MiB, deploying 256 instances using DeviceMapper took about 600 MiB in disk storage, which is about 2 MiB per container (which is more than the 500 kiB storage per VM using a backend file or snapshot). Using AUFS, the start-up time per container is below 1 second per container, whereas for devicemapper, the start-up time is about 2 seconds per container, or about 2 minutes in total for all 256 instances. In the earlier version of Docker, starting containers was done sequentially by the Docker daemon, which is a clear disadvantage compared to LXC or KVM, where all available cores can be used in parallel. Stopping all containers took even twice the time, and also all sequentially. Consequently, in the earlier versions of Docker, it was not possible to rapidly deploy huge amounts of Docker instances of the same container instances, which was definitely counter-intuitive w.r.t. the promise of lightweight containers for being lightweight and agile.

W.r.t. the memory usage, each idle container image consumed about 4 MiB of RAM. So in theory, 2k instances should only use about 10 GiB, not even taking into account KSM. Another limitation is that a standard Linux bridge only has up to 1024 ports, meaning only up to that amount of containers can be attached to the same bridge. Other solutions like OVS do not have such constrained limitation.

3.5.4 Performance Study

All results in this section were done on the more recent test platform, using a recent version of QEMU and Docker. On this platform, we studied the start-up time, memory consumption and CPU overhead, as well as done some performance tests when triggering all OpenWRT instances from a remote server. For Docker, we used the OverlayFS overlay driver available in kernel 3.16, which does not have the bug of the AUFS driver nor the 256 instances limitation of DeviceMapper.

3.5.4.1 Start-up time

In Figure 66, the average launch and boot time is depicted for deploying a particular number of VM or container instances in parallel. By launch time, we mean the overall time needed to create all new VM or container instances on the host, but excluding the internal booting time of each instance. By boot time, we refer to the total time needed before all launched instances are completely bootstrapped and are up-and-running. In case of the VMs, this also means booting the guest OSes.

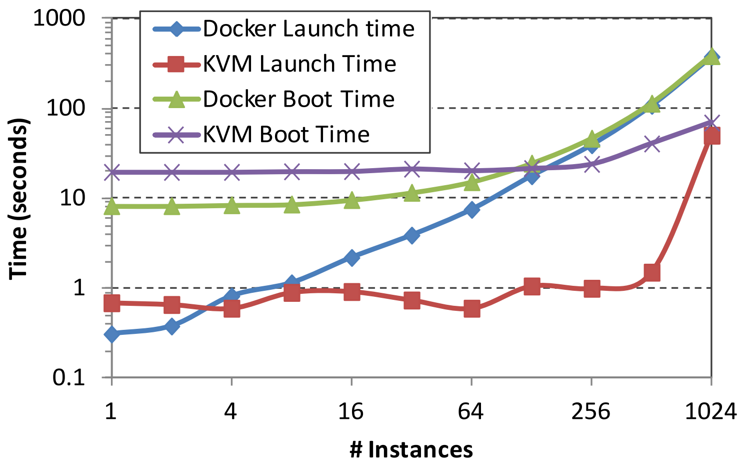


Figure 66 – Launch and boot time of Docker and KVM as a function of the number of instances.

A number of things can be observed. First, the launch time of KVM VMs is roughly constant (up until some threshold), whereas for containers, the launch time is almost linear. Secondly, due to the relatively high boot time of an operational OpenWRT instance (i.e., about 10 seconds), the guest OS boot time of the VMs only doubles the total boot time for a small number of instances. In case of a true micro-service deployment, the relative difference in boot time between a Docker instance and a VM instance would be much higher due to the relative high guest OS boot time overhead. More interestingly even, for a large number of instances being deployed (at the same time), the total boot time of KVM VMs is smaller than those of the Docker container instances (i.e., at around 128 instances). This is caused by the linear launch time overhead of Docker containers that starts to dominate the total boot time.

3.5.4.2 Memory Utilization

As depicted in Figure 67, there is a 1+ order of magnitude difference in total memory utilization when comparing VMs versus containers. Note that for KVM, we did not enable KSM, which typically results in an up to 2x reduction in memory usage.

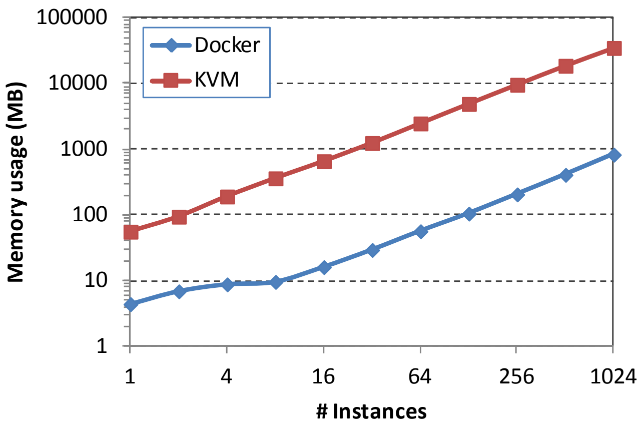


Figure 67 – Memory utilization of Docker versus KVM OpenWRT instances.

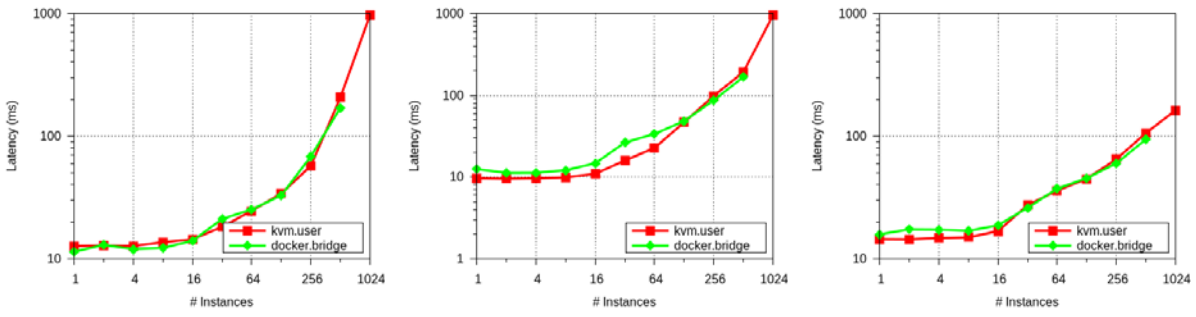
For full OS VMs such as VMs containing an Ubuntu or Centos guest OS, the expected difference in memory utilization probably will be even higher (i.e., 2+ order of magnitude). In case of unikernels on the other hand, it will be smaller, though there will always be the memory fragmentation issue in that you need to preprovision at boot time the maximum amount of memory to allocate to each VM.

3.5.4.3 CPU utilization

We performed two types of tests on the new test platform with the more recent host OS kernel, QEMU and Docker versions. In idle mode and with 1024 running VM or container instances, the system is about 99.8% idle in case of 1024 active Docker containers, and about 98.4% idle in case of 1024 active KVM VM instances on the Xeon system.

To evaluate the performance under load, we evaluated the roundtrip latency for triggering the uhttpd service within each OpenWRT instance by fetching the main web page. These requests were triggered from another machine. We experimented with different request rates and number of parallel clients.

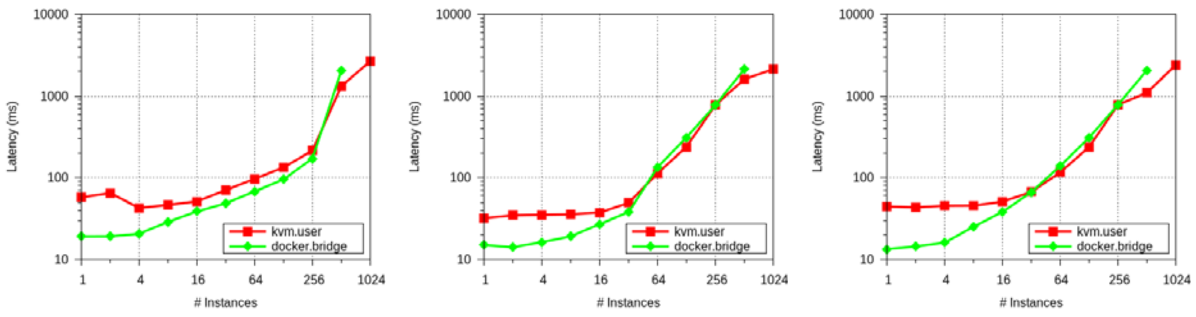
In Figure 68, the average HTTP GET roundtrip latency results are depicted for fetching the home page of each running instance using a single requesting client per instance at three different rates. As can be observed, the difference between KVM and Docker using their default networking is negligible for any request rate and for any number of parallel instances.



(a) Request rate = 1/s (b) Request rate = 10/s (c) Request rate = 100/s

Figure 68 – Average HTTP roundtrip latency for fetching a web page from each running instance using a single requesting client per instance at a request rate of 1/s, 10/s and 100/s, respectively.

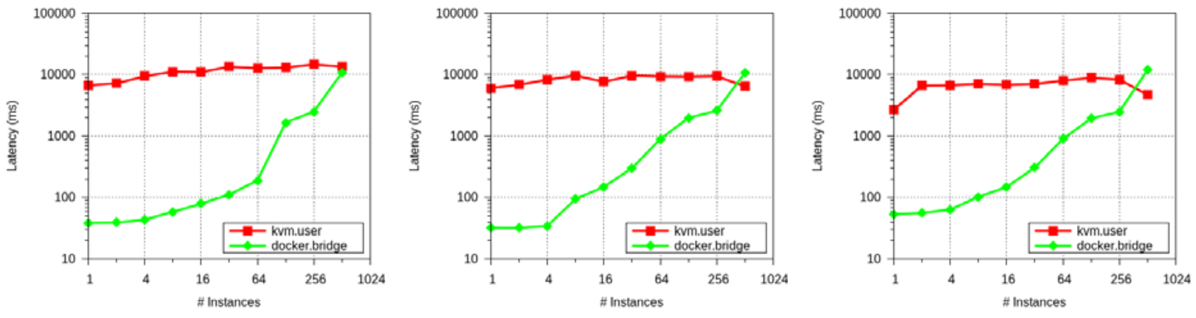
In Figure 69, we subsequently increase the number of parallel clients per instance to 4. In these graphs, we start to see already a different in roundtrip latency for a small number of parallel instances. For a large number of instances, the system gets saturated and starts dominating the latency results, nullifying the performance benefits of containers over VMs.



(a) Request rate = 1/s (b) Request rate = 10/s (c) Request rate = 100/s

Figure 69 – Average HTTP roundtrip latency for fetching a web page from each running instance using 4 parallel requesting clients per instance at request rates of 1/s, 10/s and 100/s, respectively.

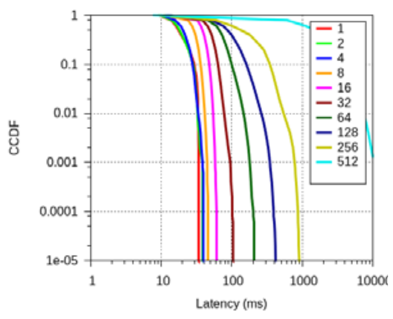
When further increasing the number of parallel clients, the performance difference increases. In Figure 70, we show the latency results for 16 parallel clients per OpenWRT instance making requests to all instances at a particular target request rate.



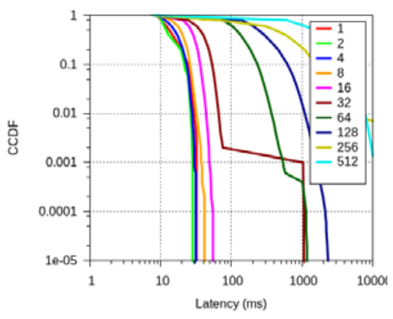
(a) Request rate = 1/s (b) Request rate = 10/s (c) Request rate = 100/s

Figure 70 – Average HTTP roundtrip latency for fetching a web page from each running instance using 16 parallel clients per instance at request rates of 1/s, 10/s and 100/s, respectively.

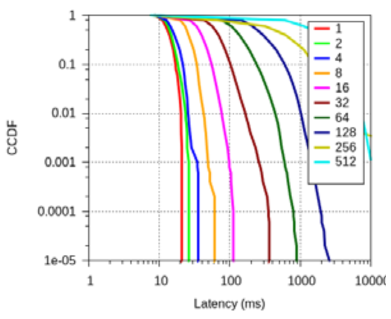
As a final result, we also show the tail latency behaviour in Figure 71 of the test case with 4 parallel requesting clients per instance by presenting the CCDF curves for each scenario. The upper graphs show de results for Docker containers, whereas the lower graphs show the results for KVM VMs. The different curves in each graph represent the behaviour for a particular number of OpenWRT instances. As can be observed, the tail latency of Docker instances is significantly better on average than those of KVM instance, especially at the lower number of instances. For the higher number of instances, the differences are much smaller. Notice also the worse tail latency behaviour of KVM for a request rate of 1/s compared to 10/s. This may be explained by a higher likelihood that a particular VM is still active when a second request comes in, whereas at low frequencies, a particular VM may have to be scheduled first on the hypervisor/host.



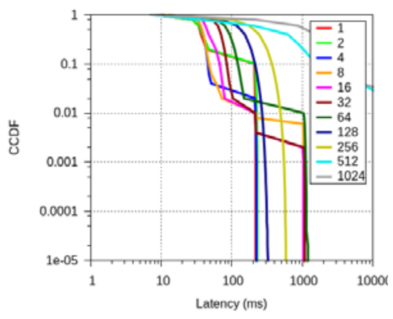
(a) Request rate = 1/s



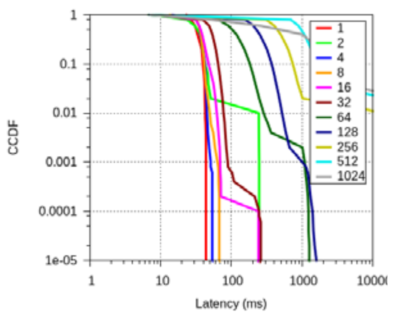
(b) Request rate = 10/s



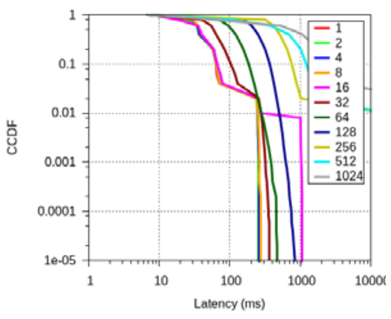
(c) Request rate = 100/s



(a) Request rate = 1/s



(b) Request rate = 10/s



(c) Request rate = 100/s

Figure 71 – CCDF curves showing the tail HTTP roundtrip latency behaviour for fetching a web page from each running instance using 4 parallel clients per instance at request rates of 1/s, 10/s and 100/s for Docker (upper graphs) and KVM (lower graphs), respectively.

3.5.5 Conclusions

In this section, we presented the results from a case study that we have done regarding extreme service density for control-plane services, and we investigated and compared the behaviour and impact of deploying these services as full VMs versus lightweight containers. We also evaluated the impact of various optimizations to further improve the overall density, both on a classical cloud server as well as on a microserver. As described, we came across various issues and problems that had to be circumvented or solved in order to reach high densities. Performance-wise, there was a clear difference in QoS (i.e. roundtrip latency) in case of multiple parallel requests, but only up to a particular amount of load, after which the performance/QoS difference decreases.

3.6 Heterogeneous Cloud Platform

3.6.1 High-Level Architecture & Design Considerations

Throughout this chapter, we repeatedly stressed the importance of a dynamic and flexibility heterogeneous cloud platform for efficiently managing heterogeneous and demanding cloud applications on a heterogeneous set of hardware and software environments. In a truly distributed and dynamic heterogeneous cloud environment, application providers cannot know up-front what the capabilities will be of the execution nodes onto which their services will be deployed (e.g., close to the user), and also vice versa, the heterogeneous cloud providers do not know either up-front the specific requirements of the demanding applications.

In contrast, in a centralized IT cloud, cloud providers typically offer a limited set of *flavours* or *instance types* (in this Deliverable called *runtime environments*) that work well for most of the classical IT cloud services. Service providers on the other hand have a relatively good overview about the specifications and possibly runtime characteristics (e.g., if they do some manual profiling on the static set of runtime environments) of the limited set of runtime environments that are being offered in a specific cloud. Unfortunately though, this knowledge is typically not 1:1 transferable to similar runtime environments from other cloud providers, especially for very demanding services.

As such, in a dynamic distributed environment such as FUSION, where application providers can register demanding applications with very different requirements, and where also third-party cloud providers could provide DCs, DCAs or execution zones to one or more FUSION domains, we need a very flexible and automated ecosystem that nevertheless provides the necessary performance, QoS, and efficiency for both parties.

Top-down, the FUSION architecture already provides a set of enabling mechanisms and features to be able to deal with this in a flexible and dynamic way. Apart from the multi-layer orchestration architecture and the various high-level placement algorithms, key concepts such as evaluator services and session slots provide crucial and abstract information from the application running in a particular low-level runtime environment in some remove data centre.

Bottom-up, from a heterogeneous cloud platform (HCP) perspective, key design considerations include the following:

- **Automated**
  The HCP should be fully automated, allowing any type of application to make optimal use of the available HW & SW resources, and vice versa.
- **Flexible**
  The HCP should be flexible enough, having a wide range of HW & SW capabilities and configuration mechanisms at its disposable, for dynamically partitioning and configuring HW & SW platform.
- **Dynamic**
  The HCP should be able to dynamically adapt to changing runtime conditions, and take necessary (corrective) actions when needed, to ensure that the (critical) applications are not impacted.
- **Optimized**

  The HCP should continuously try to optimize the runtime of both existing as well as new applications, by learning from previous deployments about what combinations of HW resources, SW configurations and application combinations work best, and which do not work well, e.g., due to too much noisy interference or other bottlenecks.
- **Robust**

  Last but not least, the HCP should be robust, not being too aggressive when applying particular optimizations, to avoid impacting the QoS of critical applications, but rather being conservative in the application of particular optimizations or configurations. If a training phase is possibly (e.g. via evaluator services), then different optimizations could be evaluated more aggressively, without impacting the actual applications. In case of issues, a self-repairing module could also be included, trying to revert to a previously known working configuration with minimal damage to the running applications. In case of configurable micro-server architectures, this could mean dynamically adding extra compute, memory, networking or storage capacity.

- **Informed**
  To be able to properly implement all steps above, the HCP needs to be very well aware of the static and dynamic runtime behaviour of all resources, software platform and application services. As such, proper active and passive monitoring and profiling tools should be available to measure, as well as modules for evaluating the overall state as well as the specific state of each component.

A conceptual high-level picture of how we envision an automated heterogeneous cloud platform should operate, is depicted in Figure 72. From a high-level point of view, we foresee three key steps that need to be integrated with each other in a continuous optimization cycle:

- Capture the requirements of applications and services through offline and online profiling and monitoring;
- Characterize the static and dynamic capabilities and constraints of the various resources, hardware infrastructures and platforms;
- Optimize the placement and configuration of the different workloads on the available execution environments.

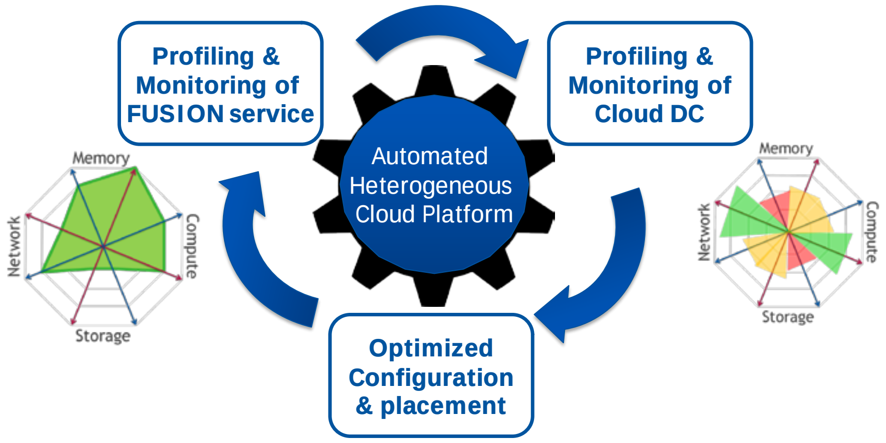

Profiling & Monitoring of FUSION service
Profiling & Monitoring of Cloud DC
Automated Heterogeneous Cloud Platform
Optimized Configuration & placement
Memory
Compute
Storage
Network
Memory
Compute
Storage
Network

Figure 72 – Conceptual view of a Automated Heterogeneous Cloud Platform.

These design considerations in mind, an architectural diagram of a heterogeneous cloud platform where continuous monitoring, profiling and optimization is at the core, is depicted in Figure 73.

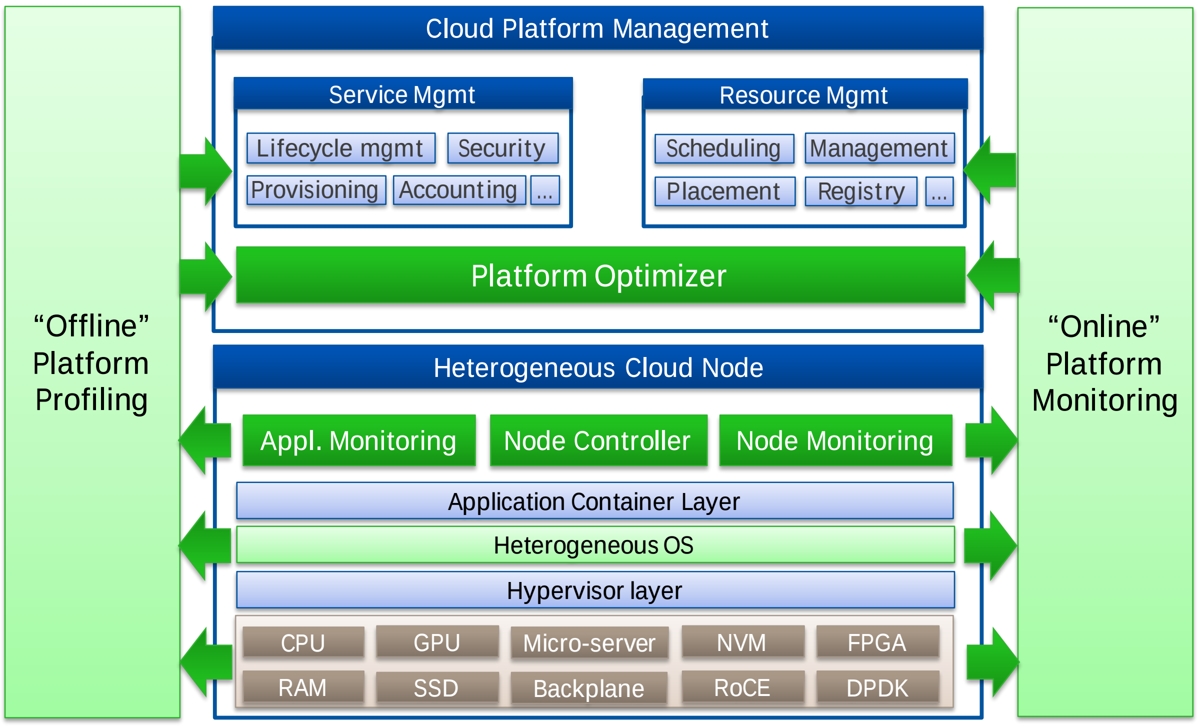


Figure 73 – High-level architecture of a Heterogeneous Cloud Platform.

Apart from the usual cloud infrastructure and platform components (shown in blue) we expand upon existing and add additional components (shown in green). These additional or extended (green) components implement the functionality of an automated continuous self-optimizing heterogeneous cloud platform, based on monitoring and profiling of both the applications as well as hardware and platform resources.

This can entail both automatically extracted information as well as information explicitly provided by the application and/or hardware/software resources. For example, application session slots, possibly in combination with additional abstract application QoS/QoE information (either directly or indirectly via evaluator services), can be combined with other information to have detailed view and insight in how efficient an application is running in a particular runtime environment. Below a short description of the key additional and/or extended components:

- **Heterogeneous OS**

  Rather than using a single OS for all nodes, in a truly heterogeneous cloud environments, different OSes could be better suited for particular hardware nodes. For example, in case of dynamically reconfigurable microservers, deploying a vanilla Linux OS distribution may be counterproductive. Also, in some cases, if may be beneficial to run a truly real-time OS, amongst other OSes, or standard OSes, but with specific kernel configurations, versions and settings.

- **Application Monitoring**

  On the nodes itself, an application monitoring probe may be deployed, collecting both information coming from the applications directly, as well as measured automatically by relevant resource accounting (OS) tools, syscall profiling, performance counter tools and others. This information is sent to the offline and/or online platform monitoring and profiling services.

- **Node Monitoring**

  Apart from the application specific monitoring tools, overall monitoring tools that monitor the various resources and OS statistics, are also collected by a series of node monitoring tools, aggregated per node, and forwarded to the platform monitoring/profiling services.

- **Node Controller**

  A key component is the node controller, which is responsible for reconfiguring, tuning and controlling a node (i.e., both the HW resources as well as software platform and OS). This controller will ensure that the proper (performance) sandboxing mechanisms, real-time guarantees, OS and resource configuration options are properly set, both for the entire node as well as for individual applications. The addition of new applications may also require some adjustments in the configuration of existing applications to reduce noisy neighbour behaviour, scheduling or other issues.

- **(Offline) platform profiling**

  In order to have an accurate (high-level) profile of the characteristics and capabilities of particular applications and particular runtime environments and execution nodes, a profile is generated and/or updated whenever a particular application is being deployed. In case of a node profile, this profiling could be a combination of offline benchmarking combined with actual monitoring data from actual service deployments, combining the results from both active and passive monitoring of a node. In case of application services and evaluator services, we can also envision both modes of operation.

- **(Online) platform monitoring**

  Live runtime statistics from both the application services and execution nodes are collected by this component, processed, and forwarded to both the general cloud orchestration layer (e.g., for accounting and high-level placement) as well as the heterogeneous platform optimizer.

- **Platform optimizer**

  The responsibility of this crucial component is to optimize how execution nodes are being (re)configured and how applications are being deployed on those optimized execution nodes.

In the next section, we will refine this high-level architecture into a high-level design.

3.6.2 Design

A high-level design, focussing on the lower parts of the architectural drawing in Figure 73, is depicted in Figure 74. On the left side of the diagram, the offline profiling portion is shown. Via offline metrics and possibly specific node and application benchmarks, application and node profiles are constructed and combined into so-called templates.

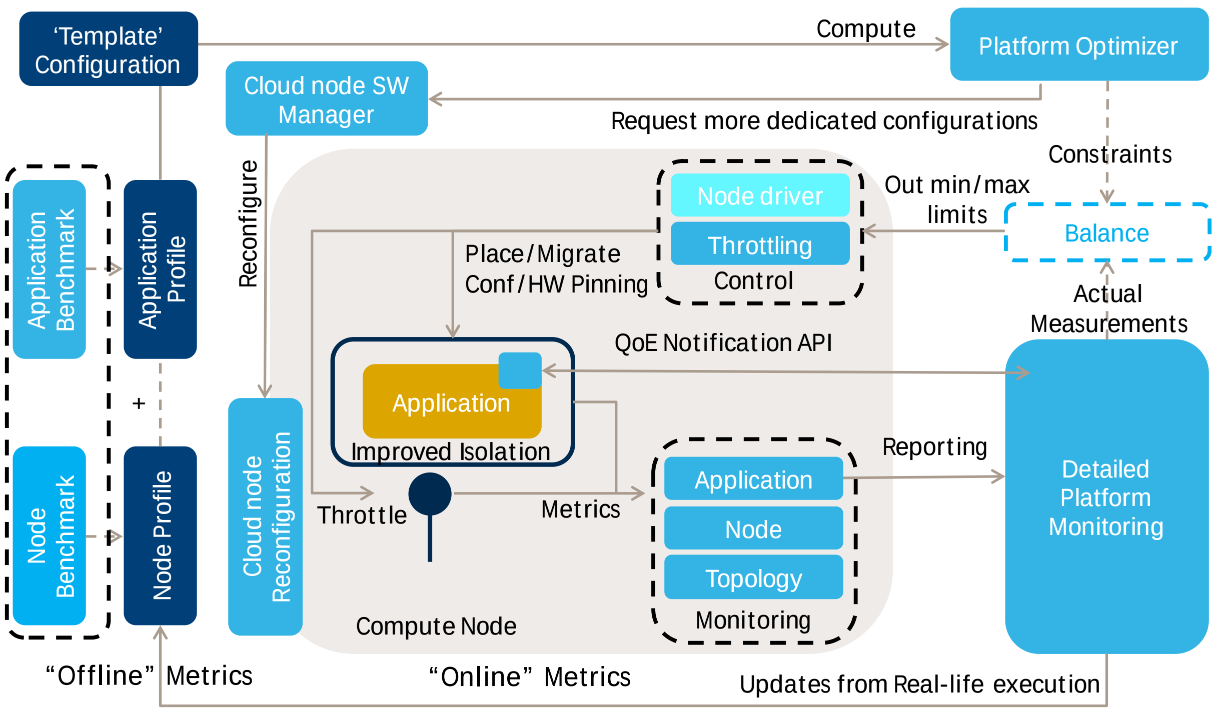


Figure 74 – High-level design of a Heterogeneous Cloud Platform Node.

From these templates, the platform optimizer will compute the optimal node, as well as how the cloud node may need to be reconfigured. The platform optimized will also set necessary resource constraints so that the application (and any other already running applications) will run properly on the execution node. These constraints are balanced with the actual live monitoring data coming from these nodes and forwarded to the node controller agent that will set and throttle the application(s) appropriately.

Application, node and topology metrics are being captured and collected, along with some QoE notification API so that the application can return feedback on how well the application is running on the constrained runtime environment. One application QoE metric includes the session slot availability, but others can be included as well, e.g. regarding the application response latencies, deadline misses, etc. With this, performance-predictability curves can be constructed, from which both a profile as well as better node configuration settings can be determined in long term. In short term, this can also result in immediate rebalancing of resources, throttling particular applications less or more, as needed.

3.6.3 Prototype Implementation

Parts of the high-level design were developed and described earlier throughout this chapter as well as already in Deliverable D3.2. Other parts of the high-level design were implemented in the prototype DCA implementation, and evaluated in the end-to-end prototype in WP5 (see Deliverable D5.3).

4. FINAL DESIGN & IMPLEMENTATION

4.1 Final architecture and design considerations

The final high-level FUSION orchestration architecture and design considerations have not changed that much since the previous Deliverable, with the following key exceptions:

- Support for hierarchical orchestration
- Support for on-demand service provisioning and deployment

We will come back to these additional architecture requirements later in this Section. Most of the architectural functional blocks however have remained the same. In summary, a high-level view of the FUSION (orchestration & management) architecture, and their key interactions, are depicted in Figure 75:

- **A top-level FUSION domain orchestrator** (D), managing all registered FUSION services across either a set of distributed execution zones or resolution domains.
- **An intermediate FUSION resolution domain orchestrator** (RD), managing a subset of all registered FUSION services across a subset of all distributed execution zones, typically confined within some geographical region.
- **A FUSION execution zone** (Z), managing deployed FUSION services in a execution environment.
- **A FUSION data centre adaptor layer** (DCA), an optional adaptor layer, abstracting the details of the lower-level orchestration and management layer of the underlying data centre(s).
- **A (third-party) cloud platform or data centre** (DC), which may be managed by other entities, and which provides the necessary physical and/or virtual resources and low-level management layer for deploying all FUSION application and orchestration services.
- **A FUSION service resolver** (R), providing optimal service selection for client service requests.
- **A FUSION evaluator service** (EVAL), providing application-specific feedback for optimal service placement.
- **A FUSION application service** (EPG), leveraging FUSION architecture for providing excellent QoE towards clients.
- **A FUSION service provider** (blue avatar), registering and monitoring FUSION application services.
- **A FUSION client** (green avatar), connecting to FUSION application services.

In this Figure, we included the DCs and the fact that one top-level domain orchestrator is responsible for managing multiple intermediate resolution domains, which in its turn is responsible for managing one or more execution zones.

Also, in this and all following diagrams, unless otherwise stated, the direction of the arrows shows who is making a (RESTful) service request to what other component (e.g., a FUSION client makes a service resolution request to a service resolver (R). The response of these requests obviously go back in the other direction; however, for clarity, we excluded these return arrows in the Figures (though they are in principle always there). The black and red lines depict key FUSION protocol messages across the various layers; the red lines show new protocol interactions, related to the on-demand service deployment. The grey dotted line depicts a service-specific interaction.

The motivation and various integration models of a DCA layer with one or more execution zones and/or data centres has been discussed already in detail in Deliverable D3.2, Section 3.1, so we will not come back onto this design decision.

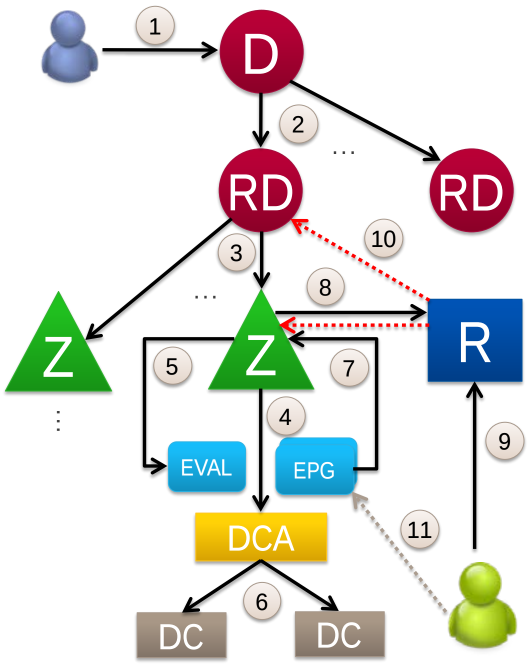


Figure 75 – Final high-level FUSION architecture diagram with key interactions.

Key interactions between the various inter-layer components have been numbered in Figure 75. Note that these interactions do not have to occur in this order or that all interactions need to occur every time; they merely represent some key interactions that will often occur when running a FUSION system:

1) A FUSION application service provider registering, deploying and monitoring application services in a global top-level FUSION domain.
2) A domain orchestrator, triggering one or more intermediate resolution domains for deploying/managing FUSION services.
3) A resolution domain orchestrator, triggering execution zones for registering/deploying/managing FUSION application and evaluator services.
4) An execution zone, triggering the DCA layer to deploy/terminate running instances, query the available runtime environments, etc.
5) An execution zone, triggering one or more evaluator services to assess the capabilities of a particular runtime environment for a particular service deployment request.
6) The DCA, triggering the actual data centre APIs for managing the actual runtime environments (e.g., physical or virtual machines) and deploying actual running FUSION instances within those (e.g., containers).
7) A FUSION application service, returning available session slot information updates to the zone manager.
8) An execution zone, pushing session slot updates to a FUSION resolver.
9) A FUSION client (could also be another service), making a service resolution request, to discover a close running instance.
10) A resolver, triggering the domain orchestrator or execution zone to deploy an instance on-demand.
11) A FUSION client, connecting to an actual instance via a service specific protocol (out-of-scope for FUSION).

Note that this only covers a key fraction of all interlayer interactions; an overview of all protocols and interactions that have been defined for FUSION orchestration, can be found in Internal Report I2.1 (Final Specification of FUSION Interfaces).

We will now come back to the two additional architectural features that were included in the final architecture of the FUSION orchestration layer, namely hierarchical orchestration and on-demand service deployment.

4.1.1 Hierarchical orchestration

We have studied hierarchical orchestration, and specifically its impact on service placement, already in Section 2.1. In that model, we started from a single global FUSION orchestration domain, and divided this into several resolution domains for better scalability. This additional (optional) layer has been included to cover the case of a world-wide global domain where application providers can register their services. The global domain will then ensure that the services are optimally deployed across the globe.

Rather than having this global domain to have to govern all execution zones across the globe (which easily could be in the 10s or 100s of thousands of zones), we decided to include an intermediate domain, which is responsible for managing a geographical subset of execution zones. One could expect to have a few of these resolution domains per continent. These resolution domains then control and manage the execution zones. Note that protocol-wise, a resolution domain could implement the same protocols and APIs as a regular execution zone, possibly with some additional features in the message format to efficiently take into account the hierarchical nature.

This has the advantage that a domain does not explicitly need to be aware of the fact that it is governing a resolution domain or an execution zone, and consequently, any number of intermediate resolution domains could be included in between the top-level domain and the bottom-level execution zones (which are responsible for the actual services). In this model, the intermediate resolution domains can be seen as an aggregated execution zone from the perspective of the higher-level domain, or domain proxy from the perspective of an execution zone, so that a domain does not have to govern all execution zones individually, or that execution zones do not have to govern a wide range of largely distributed data centres.

At the lowest layers, we envision that a single execution zone and/or DCA can be responsible of a (limited) number of distributed data centres. Especially in case of tiny edge data centre environments that only comprise a few execution nodes, having individual execution zones per node governing only a few session slots of a handful of service instances would be too wasteful. Instead, a single execution zone and/or DCA layer could easily cover a number of these. A key change then in underlying assumptions is that different runtime environment types coming from the same execution zone could be at different geographical locations, something that needs to be taken into account during placement as well as service resolution.

4.1.2 On-demand service deployment

In a highly distributed cloud environment such as FUSION, it is unrealistic to assume that sufficient service instances can be deployed in a cost-effective manner across all execution zones with a high enough utility function (e.g., roundtrip latency), unless for the most popular and frequently used services. Specifically for the long tail of services, it is simply not feasible to accurately predict the expected demand in very specific small geographical regions and predeploy sufficient instances (or even a single instance) in all relevant execution zones.

For these services, an on-demand service deployment scenario is much more beneficial, provided that new instances or session slots can be deployed in a reasonable amount of time (e.g. a few seconds maximum). To support these use cases, we investigated and implemented a number of new features, as discussed in Section 0, 3.2 and 3.3.6.3. These features include reducing the start-up latency for deploying new instances, by leveraging a lightweight application packaging and deployment model, smartly preprovisioning essential portions of all applications across the various execution zones and nodes, as well as providing the resolver insight in the amount of time it would take to provide a new session slot, by leveraging an additional queuing metric next to session slots.

To further reduce the waiting time for a requesting client, a resolver can proactively trigger the selected execution zone (e.g., one with a sufficiently low announced session queuing time and that has a good utility function) to already start provisioning and/or deploying a new session slot for a particular service, so that when the client tries to connect a few moments later, the on-demand provisioning is already being executed. As such, we need an API between the resolver and execution zone to support this case. In the general case, for when the session queuing time metric is not supported or no feasible execution zone can be found, the resolver can delegate the request to its designated (resolution) domain, who is then responsible for quickly selecting and deploying a new instance in a particular zone, and returning the location back to the resolver.

4.2 FUSION Domain Orchestrator

4.2.1 Final Design

The final version of a FUSION domain orchestrator design is depicted in Figure 76. Compared to the graph in Deliverable D3.2, key changes include the addition of the resolution domain, an explicit security component for authentication and authorisation, and the correction of some of the internal arrows. The key design considerations remained unchanged:

- **A common public interface**, shared by all external entities, and protected by proper user/role based authentication and authorisation mechanisms.
- **A modular decomposition** of the key domain orchestration functions, each with their own well-defined RESTful interfaces.
- **State, configuration and runtime data** is maintained in highly available distributed key value store such as ETCD, facilitating the implementation and management of a distributed modular domain orchestrator.
- **Scalability and high availability** of each of the functional components, possibly deployed on top of the execution zones that are being governed.

Note that this diagram can represent both a top-level domain orchestrator as well as an intermediate-level resolution domain orchestrator, as most functional blocks need to be in either orchestrator, though with slightly different implementations. In general, each domain could govern a set of both lower-layer domains as well as execution zones. Application service providers typically only will be able to see and/or access the top-level domain orchestrator, unless a domain manager specifically wants to allow application providers for registering and deploying only within regional domains.

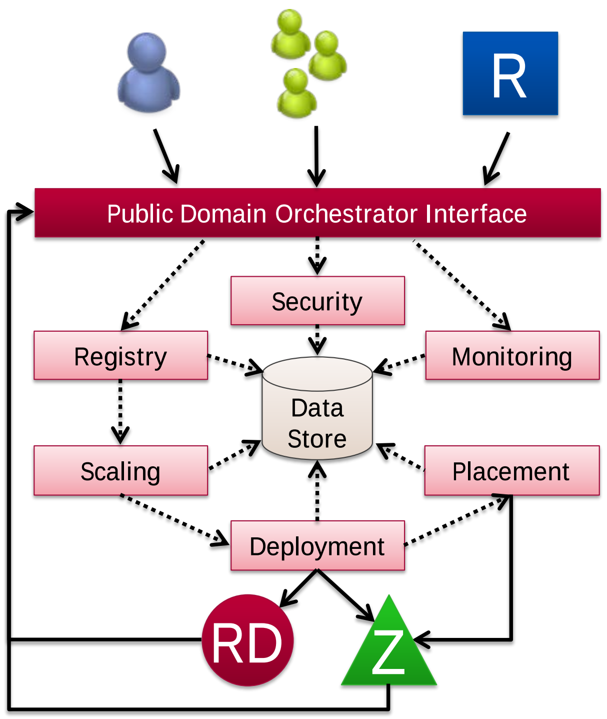


Figure 76 – Final high-level design of a FUSION domain orchestrator.

As a reminder, in these diagrams, the direction of the arrows indicate which component is making a (RESTful) service request to what other component; the response message is not shown, but obviously is present in all cases. The full lines represent inter-layer protocols and interactions, whereas the dotted lines represent internal protocols. Note that this is only one example of how a FUSION domain orchestrator can be designed, and that other (simpler or more complex) designs as possible. Note also that this diagram does not reflect the fact that these components in practice for scalability and reliability typically will be replicated into a number of instances that all need to be kept in sync with each other. For interoperability, only the public inter-layer protocol interfaces and their expected functionality are essential. In FUSION, we focused on the latter two, both through experimentations/simulations as well as a simplified but functional prototype.

A brief summary of the key functionality of each of these components is presented below. More details on the concepts and algorithms behind these components, as well as their basic implementation and evaluation, can be found throughout both this Deliverable and Deliverable D5.3.

- **Public Interface**

  This is the public and secure (https) API service that implements the public interface and delegates the API calls to the internal components.

- **Data Store**

  This is a highly available data store where all components can read and/or write their state, monitor and/or wait for state changes from other components, etc.

- **Security**

  This internal component is responsible for all user and role based authentication and authorisation, for the external but possibly also for the internal components (the latter interaction is not shown in the diagram for clarity).

- **Registry**

This component handles the registration, updating and early processing and parsing of new and existing services. This typically will automatically trigger the other internal components to start deploying or rescaling the number of instances of that service in particular regions.

- **Scaling**

  Based on load patterns or an explicit deployment/termination request, this component will appropriately deploy or terminate sessions of a particular service in one or more zones. This component only is responsible for *deciding* whether and when some scaling action needs to be done.

- **Deployment**

  The role of this component is to coordinate the effective deployment and/or termination of a service in one or more zones or lower-level resolution domains.

- **Placement**

  This key component is responsible for how many session slots of each services should be created or removed in particular lower-level resolution domains and execution zones. For this, it may trigger these components for running evaluator services of new services to be deployed.

- **Monitoring**

  This component gathers and processes various types of monitoring data, coming from the lower-level domains, execution zones, network as well as services.

4.2.2 Prototype Implementation

We have implemented a fully functional basic prototype of a domain orchestrator. For this prototype implementation, we focussed on implementing and validating the key APIs and their corresponding functionality. We did however not focus on the scalability of our implementation as this was out-of-scope for this prototype, though this could be added using standard web scaling techniques if necessary.

The implementation was done in Python, and is developed on top of the Flask framework, leveraging existing modules for handling all RESTful APIs as well as basic HTTP authentication. Both the internal Flask debug application server as well as the Unicorn application server was used for wrapping our code into a functional web service. We did not focus on adding HTTPS support in our prototype, but this could easily be done using tools like NGinx. All developed service components also have been packaged as Docker containers. An example code fragment is depicted below.

In this example code fragment, we first depict how some of the administration API calls for a domain orchestrator are being mapped into the Flask framework. For the zone management API, we also depicted their implementation, supporting both a GET and DELETE function. The user/role based authentication and authorisation is handled via the *decorator* function, which in this cases verifies whether a logged in user with *domainadmin* privileges is trying to access these API calls, otherwise a forbidden response code is returned.

```
api.add_resource(DomainPingAPI, '/1.0/ping', endpoint = 'ping')
api.add_resource(DomainUsersAPI, '/1.0/users', endpoint = 'users')
api.add_resource(DomainUserAPI, '/1.0/users/<string:name>', endpoint = 'user')
api.add_resource(DomainZonesAPI, '/1.0/zones', endpoint = 'zones')
api.add_resource(DomainZoneAPI, '/1.0/zones/<string:zoneid>', endpoint = 'zone')

...
class DomainZoneAPI(Resource):
    decorators = [domainadmin_privileges_required]
    #fetch information of specific zone
    def get(self, zoneid):
        zone = zones_find(zones, zoneid)
        if(zone == None):
            return fusion_response_message(ZONE_NOT_FOUND)
        return fusion_response_message(OK, zone)
    #remove specific zone
    def delete(self, zoneid):
        zone = zones_find(zones, zoneid)
        if(zone == None):
            return fusion_response_message(ZONE_NOT_FOUND)
        return zone_delete(zone)

...
```

The prototype implementation has been extended and improved in a number of ways since the previous Deliverable D3.2:

- We implemented data persistency, so that the entire state of a domain is being synchronized in a decentralized highly available ETCD key-value store. This facilitates adding high-availability and scalability into our prototype. This was also used to evaluate the impact of latency in a distributed setup.
- We decomposed some of the key domain functionalities into separate internal components, each implemented on top of the same framework as the main prototype. We did however not implement the fully decomposed design shown in Figure 76, but rather focussed on the two most important components, namely the **Scaling** and **Placement** component, as depicted in Figure 77.
- We implemented a basic version of an automated *Scaling* components, where service instances are automatically deployed or destroyed based on the available session slots and the scaling properties in the service manifest. Also, when new zones are added (or removed) automatic scaling may be involved. An additional implementation of autoscaling was also provided (via the public domain API) for validating some of our more theoretical scaling and placement algorithms.
- We implemented two versions of a *Placement* component, namely one that just randomly selects an execution zone for deploying new service instances, and another that triggers the evaluator services in execution zones for deciding where to deploy new instances. Additional evaluations using our theoretical placement algorithms were also done via the public domain API, manually selecting feasible zones, but still relying on the evaluator service placement implementation for selecting the optimal runtime environment within that zone.
- We improved the implementation of all APIs, and specifically w.r.t. the dynamic nature of services and execution zones being registered but also being removed from a domain. We also focused on properly implementing the service session slot scaling API, so that the other partners could leverage this API for their experiments, without need to deeply integrate with our prototype.
- We implemented the new two-phase evaluator service mechanism, triggering a stage-2 analyzer from the domain orchestrator.

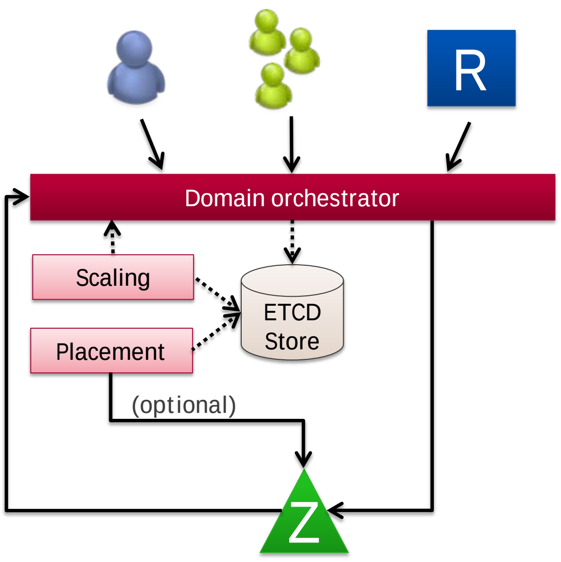


Figure 77 – High-level design of our FUSION domain orchestrator prototype.

This prototype implementation is used for the system evaluation in Deliverable D5.3.

4.3 FUSION Execution Zone

4.3.1 Final Design

The final design of a FUSION execution zone manager is depicted in Figure 78. In our model, a zone manager is not responsible for handling the lower-level details of the underlying physical or virtual infrastructure, but only is responsible for managing the high-level lifecycle management of FUSION services and instances, the provisioning of container images, session slot and stage 1 evaluator service management, multi-configuration service instances (i.e., aliasing of particular instances across multiple service types, see Deliverable D3.2), etc.

The final high-level design of a zone manager is depicted in Figure 78. From a high-level perspective the design has not change much since Deliverable D3.2 (except for the addition of the resolution domain orchestrator as well as the resolver (in case of on-demand deployment)), though the functionality of each of the major components has been refined. Just as with the design of the domain orchestrator, this is merely one example of how a generic zone manager could be decomposed. Depending on the size of the actual zone manager, some of these components may be merged, or split into a number of actual instances, for scalability and/or availability/reliability reasons.

Note again that the direction of the arrows reflects the direction of which component is making a (RESTful) service request to what other component; the full lines represent inter-layer protocols, whereas the dotted lines represent internal protocols. The grey dashed line represents a service-specific protocol between a client and a service instance. The actual service instances are being managed by the DCA layer.

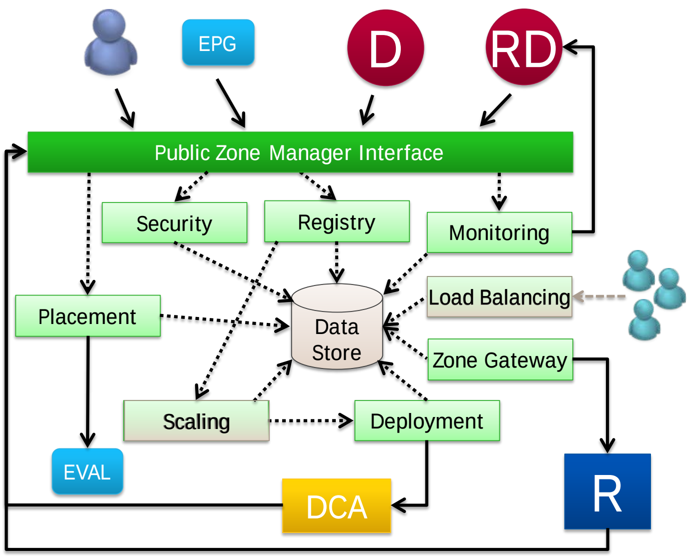


Figure 78 – Final high-level design of a FUSION execution zone manager.

A short overview of the key functionality of each of the major subcomponents, is provided below:

- **Public Interface**

  This is the public and secure (https) API service that implements the public interface and delegates the API calls to the internal components.

- **Data Store**

  This is a highly available data store where all components can read and/or write their state, monitor and/or wait for state changes from other components, etc.

- **Security**

  This internal component is responsible for all user and role based authentication and authorisation, for the external but possibly also for the internal components (the latter interaction is not shown in the diagram for clarity).

- **Registry**

  This component handles the registration and further updates (e.g., w.r.t. autoscaling or on-demand scaling). This will ensure the service is known by the zone manager, and may trigger automatic (partial) preprovisioning of the service image layers into the zone and/or particular hosts.

- **Placement**

  This component is mainly responsible for properly managing the various runtime environments offered by the DCA layer, smartly deciding where to deploy evaluator services, and effectively triggering these evaluator service probes and generating corresponding offers (e.g., associate a particular price with a particular evaluation request).

- **Scaling**

  This component will ensure that a proper amount of (available and/or total) session slots will be present inside the execution zone. Especially in case a zone has the flexibility to autoscale, scale on-demand or e.g., provide session slots in a lazy manner, this component will monitor and if necessary trigger the instantiation or termination of instances.

- **Deployment**

This component has two functions: one is to delegate the provisioning, deployment and termination of FUSION services to the DCA layer; the second is to also trigger the allocation and deallocation of runtime environments, causing the execution zone to expand or shrink. This is especially of interest in case a FUSION execution zone is being deployed on a third party cloud or data centre, where resources are only effectively allocated when needed to save costs.

- **Monitoring**

  This component collects, aggregates and processes all types of runtime, monitoring and profiling information, and updates them into the data store, which in its turn can trigger other components to perform particular actions. For example, changes in session slot availability will typically trigger the scaling and zone gateway components to take some actions.

- **Load Balancing**

  Unless in case of stateful sessions, a requesting client typically does not care which service instance and session slot within an instance it is assigned to. Rather than forwarding session slot availability of individual service instances to the service resolution plane, a zone manager can aggregate all of them and announce them via a single public IP address to the resolution plane. In Section 2.9 of Deliverable D3.2, we already discussed how the intra-zone load balancing could be implemented using e.g. flow-based load balancers. Note that application services, as part of their composite service graph, could also provide custom load balancing via a specific load balancing service.

- **Zone Gateway**

  This component interacts with the FUSION service resolution plane, providing updates on session slot availability. The format and frequency of these updates will impact the amount of monitoring data that needs to be exchanged. In Section 3.3.6.4, we already discussed various mechanisms on how to minimize this.

4.3.2 Prototype Implementation

We have also implemented a fully functional basic prototype of a zone manager, focussing on the functionality of the evaluator services, session slot management, and interaction with the other FUSION layers (DCA, application services, resolution plane and (resolution) domain orchestrator. Again, we did not focus on the scalability of our implementation as this was also out-of-scope for this prototype; existing high-availability and scalability techniques could easily be adopted and integrated if needed. The main purpose of this prototype is to validate the key functionality of a zone manager in the context of a full FUSION platform.

The implementation was done using the same software platform as for the implementation of the domain orchestrator prototype, namely the Flask Python framework as well as additional modules for handling the RESTful APIs and HTTP authentication, and wrapped as a Docker container. Since the previous Deliverable D3.2, the zone manager prototype has been extended and improved in a number of ways:

- We implemented data persistency, so that the entire state of a zone manager is being synchronized in a decentralized highly available ETCD key-value store. This facilitates adding high-availability and scalability into our prototype. This was also used to evaluate the impact of latency in a distributed setup.
- We improved the implementation of all APIs, and specifically w.r.t. the dynamic nature of services and DCA(s) being added but also being removed from a zone. We also focused on properly implementing the service session slot scaling API within a single zone, to properly implement various scenarios as described in Deliverable D5.3.

- We added a client-ping API, which is triggered by the resolver, and allows a zone to ping the publicly addressable IP-address of a client, in order to have a rough estimate of the latency between a zone and a client. This was used in our greedy FUSION service resolver prototype implementation.
- We added the fast-track on-demand service provisioning and deployment APIs, which can also be triggered by the resolution plane, for triggering a zone to quickly increase the number of available session slots for a particular service.
- Support was added to be able to handle the heterogeneous set of runtime environments, exposed and provided by the DCA layer. This includes deployment of evaluator services on (a subset of) the available runtime environments, checking for particular static requirements in the manifest to be able to pre-filter the runtime environments, etc.

This prototype implementation is used for the system evaluation in Deliverable D5.3.

4.4 FUSION DC Adaptor

4.4.1 Final Design

The goal of a FUSION DC Adaptor (DCA) is to hide the complexities and specificities of an underlying heterogeneous and possibly third-party cloud environment or data centre from the core functionality of a zone manager (which is mainly about managing application and evaluator services and session slots). In Deliverable D3.2, we proposed three different DCA designs, reflecting three different levels of integration with the actual underlying data centre:

- A *physical DCA*, where the DCA has direct access and control to the physical compute and networking infrastructure.
- A *virtual DCA*, where the DCA needs to map all FUSION functionality onto the APIs of the existing underlying cloud environment, such as for example Amazon EC2 or OpenStack. In this model, it only can configure the virtual infrastructure.
- A *hybrid DCA*, where the underlying cloud environment provides particular FUSION-aware accelerators and/or APIs that improve the overall efficiency of the DCA implementation. This is a similar technique to a paravirtualization in virtualized environments.

A representative design of each of the variants is depicted in Figure 79. More information on the functionality and relationship between components can be found in Deliverable D3.2, Section 3.4.1.

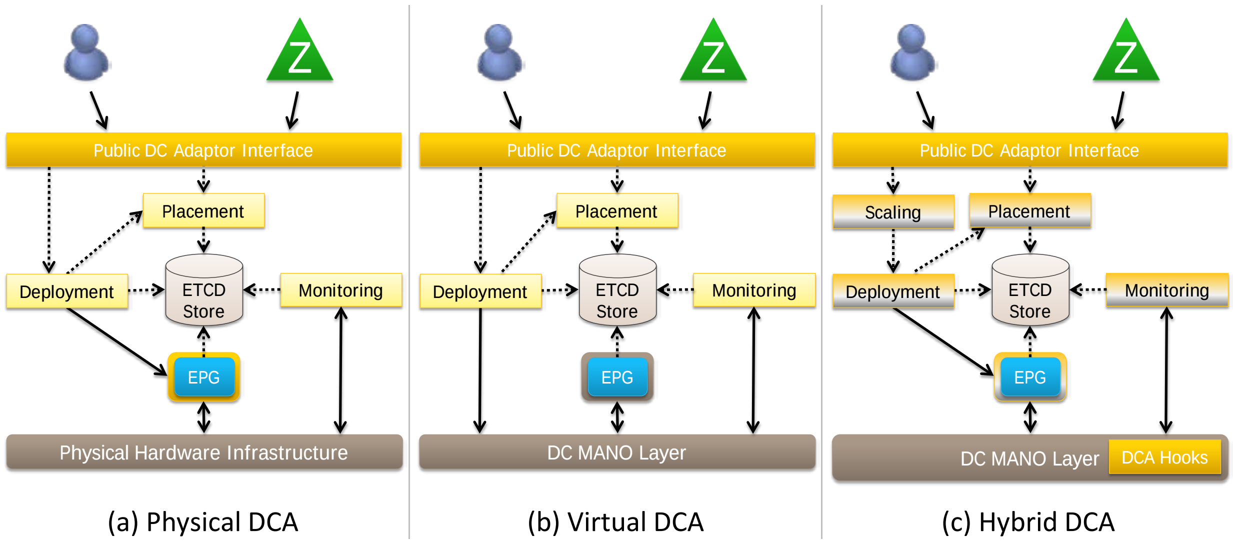


Figure 79 – Final high-level designs of physical, virtual, and hybrid DCA layer.

4.4.2 Prototype Implementations

We provided an implementation of each of the three options. In case of the hybrid design option, we already provided a basic implementation of a FUSION session slot based scaling mechanism by adding and integrating extra FUSION-aware capabilities in an OpenStack environment. This was already described and evaluated in Deliverable D3.2.

In case of the virtual design option, we implemented a flexible adaptive DCA layer on top of our private OpenStack cloud environment, automatically allocating/deallocating, provisioning and specializing VMs of particular flavours, based on the requirements and load. For this, we integrated our Python prototype with the Python OpenStack APIs.

The most extensive implementation of the DCA, implementing all key FUSION APIs, and which is being used for all system evaluation scenarios in D5.3, is based on the physical design option, where we have most impact and control on the underlying physical hardware, providing the most flexibility and reliability for implementing and evaluating key features of our heterogeneous cloud platform.

The core implementation(s) of the DCA service were also done in Python, leveraging the same Flask and Restful frameworks and modules for implementing the RESTful APIs. Compared with previous versions of the DCA, we expanded and improved the implementation in a number of ways:

- We integrated with existing container clustering frameworks such as swarm and mesos for automatically deploying constrained containers across a set of constrained (physical and/or virtual) nodes.
- We provided support for services packaged as Docker containers, as well as services packaged as KVM virtual machines as well as unikernels.
- We integrated with OpenStack APIs in case of the virtual DCA implementation.
- We integrated the Rancher framework for monitoring and displaying basic host monitoring data.
- We implemented the concept of multiple types of runtime environments, managing available nodes and deploying containers in different manners. Example runtime environments include best-effort versus real-time guaranteed environments, GPU-enabled environments, environments with a large vs small amount of memory, cores or CPU quota, etc.
- We implemented two ways in which containers are visible from outside, one via NAT, and another via bridging (and possibly floating IPs).
- We integrated our implementation in the vWall environment as well as the jFed environment.

This prototype implementation is used for the system evaluation in Deliverable D5.3.

5. SUMMARY

In this deliverable, we provided the final version of the FUSION service orchestration and management plane, focusing on the key new contributions w.r.t. algorithms, design and implementation of each of the key components.

We started this deliverable by motivating the relevance and need for a FUSION-alike distributed heterogeneous cloud platform for managing demanding interactive (real-time) services from the cloud, followed by a recap of all key FUSION concepts and contributions.

Next, we presented a set of new algorithms for handling placement in a hierarchical orchestration framework, and in case of composite services. We also evaluated a number of algorithms to optimize on-demand service provisioning.

In Chapter 3, we motivated the need for a heterogeneous cloud platform for cost-efficiently managing these types of services, and discussed the benefits of a lightweight composite application service model. In a truly dynamic and flexible heterogeneous cloud environment, profiling and monitoring is of the essence, both from an application service as well as cloud platform perspective, to be able to efficiently cope with various levels of application and hardware heterogeneity. As such, we discussed a number of profiling mechanisms as well as a mechanism for visually representing the trade-off between *performance* and *predictability*. The two key FUSION orchestration concepts, namely session slots and evaluator services, were extended and analyzed in detail. Finally, efficient inter-service communication mechanisms were evaluated, as well as the results of an extreme node density study were presented. All this was concluded in the presentation of a heterogeneous cloud platform design that could implement the necessary capabilities in a dynamic manner.

In Chapter 4, we presented the final design of the FUSION orchestration layer, and zoomed in on the design of each of the key components. A short summary of the final state of the prototype implementation was also presented.

A list of the final inter-layer orchestration protocol specifications, can be found in a separate document, namely Internal Report I2.1 (Final Specification of FUSION Interfaces).

6. REFERENCES

[AFM16] M. Aly, M. Franke, M. Kretz, F. Schamel, and P. Simoens, "Service Oriented Interactive Media (SOIM) Engines Enabled by Optimized Resource Sharing", In Proc. IEEE SOSE 2016, *to appear*.

[ALA09] N. Alam. "Survey On Hypervisors." School Of Informatics and Computing, Indiana University, Bloomington (2009).

[Bo12] F. Bonomi et al., "Fog computing and its role in the internet of things", in Proceedings of the first edition of the MCC workshop on Mobile cloud computing. pp. 13-16, 2012.

[BRITE13] BRITE: Boston university Representative Internet Topology gEnerator. 2013.

[BRUM13] Glen Van Brummelen. "Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry" in Princeton Uni. Press, 2013.

[CPLEX] CPLEX solver. www-01.ibm.com/software/commerce/optimization/cplex-optimizer

[DCMAP] http://www.datacentermap.com/

[DATAS] https://github.com/richardclegg/multiuservideostream

[DWW05] M. Dick and O. Wellnitz and L. Wolf. "Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games" in 4th ACM SIGCOMM Workshop on Network and System Support for Games, 2005.

[Fa11] J. Famaey et al., "Network-aware service placement and selection algorithms on large-scale overlay networks", in Computer Communications, Elsevier, Vol. 34, pp. 1777-1787, 2011.

[Fi14] J. Fink, "Docker: a Software as a Service, Operating System-Level Virtualization Framework", Code4Lib Journal, Vol. 25, 2014.

[FIO14] Fio I/O benchmarking tool, http://freecode.com/projects/fio.

[Go16] Google, "Google joins Open Compute Project to drive standards in IT infrastructure", March 2016, https://cloudplatform.googleblog.com/2016/03/Google-joins-Open-Compute-Project-to-drive-standards-in-IT-infrastructure.html

[Go16b] Google, "Google Cloud Platform adds two new regions, 10 more to come", March 2016, https://cloudplatform.googleblog.com/2016/03/announcing-two-new-Cloud-Platform-Regions-and-10-more-to-come_22.html

[He16] B. Henrion, "Docker OpenWRT Image", http://www.zoobab.com/docker-openwrt-image.

[iLABT13] iLab.t Virtual Wall | Internet Based Communication Networks and Services, 2013.

[Kn11] S. Knight et al, "The Internet topology zoo", Selected Areas in Communications, IEEE Journal on, Vol. 29, pp. 1765-1775, 2011.

[LANDA13] R. Landa, J. Araujo, R. Clegg, E. Mykoniati, D. Griffin, and M. Rio, "The large-scale geography of internet round trip times," in IFIP Networking Conference, 2013, 2013, pp. 1–9.

[Li14] J. Li et al., "Lower and upper bounds for a two-stage capacitated facility location problem with handling costs", European Journal of Operational Research, Elsevier, Vol. 236, pp. 957-967, 2014.

[Me14] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment", Belltown Media, 2014, Linux Journal, Vol. 2014, p. 2.

[Mi16] R. Miller, "Facebook Takes Open Compute Hardware to the Next Level", March 2016, http://datacenterfrontier.com/facebook-open-compute-hardware-next-level/.

[NAA10] R. Nathuji, K. Aman, and G. Alireza. "Q-clouds: managing performance interference effects for qos-aware clouds." Proceedings of the 5th European conference on Computer systems. ACM, 2010.

[NS14] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture", in International Journal of Open Information Technologies, Vol. 2, pp. 24-27, 9, 2014.

[Oh16] K. Ohannessian, "The technical challenges of Virtual Reality", Tech Innovation, July 2016, http://iq.intel.com/the-technical-challenges-of-virtual-reality/.

[PW02] L. Pantel and L. Wolf. "On the Impact of Delay on Real-time Multiplayer Games" in 12th Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 2002.

[Su16] B. Sullivan, "Google Set For Massive Cloud Expansion With 12 New Data Centres", March 2016, http://www.techweekeurope.co.uk/cloud/cloud-management/google-cloud-expansion-12-new-data-centres-188443

[SYG02] A. Syarif, Y. Yun, and M. Gen, "Study on multi-stage logistic chain network: a spanning tree-based genetic algorithm approach", Computers & Industrial Engineering, Elsevier, Vol. 43, pp. 299-314, 2002.

[Va09] F. Vandeputte, "Vampire Parallelization Toolchain", IWT Vampire project, Deliverable D.B.4.3, 2009.

[VE09] F. Vandeputte, and L. Eeckhout. "Finding stress patterns in microprocessor workloads." High Performance Embedded Architectures and Compilers. Springer Berlin Heidelberg, 2009. 153-167.

[VER14] L. Vermoesen, Jean-Paul Belud, Emilio Lastra, Luc Ogena, Frederik Vandeputte, "On the Economical Benefit of Service Orchestration and Routing for Distributed Cloud Infrastructures: Quantifying the Value of Automation.", White Paper, (2014), http://www.fusion-project.eu/

[We16] M.K. Weldon, "The Future X Network: A Bell Labs Perspective". CRC Press, 2016.

[Wu15] T. Wu et al., "A lagrangean relaxation approach for a two-stage capacitated facility location problem with choice of depot size", Networking, Sensing and Control (ICNSC), 2015 IEEE 12th International Conference on. pp. 39-44, 2015.

[Zh10] Q. Zhang et al., "Dynamic service placement in shared service hosting infrastructures", in Networking 2010, Springer, pp. 251-264, 2010.

7. APPENDIX A: FINAL PROTOCOL SPECIFICATIONS

All inter-layer and intra-layer orchestration APIs have been designed based on RESTful principles, leveraging existing HTTP(S) authentication and authorization mechanisms. Although the message format of these RESTful APIs could be of any kind, we focused on designing and implementing all API commands using JSON as message format, to simplify the implementation in our prototype.

In this Deliverable, we will not provide an overview nor describe all APIs in detail. The full protocol specifications are described in a separate document, combining also the key protocol specifications of the other major FUSION layers apart from those relevant for Work Package 3. We kindly refer to Internal Report I2.1 (Final Specification of FUSION Interfaces) for a detailed overview and description of all APIs.