ECSEL2017-1-737451

FitOptiVis

From the cloud to the edge - smart IntegraTion and OPtimisation Technologies for highly efficient Image and VIdeo processing Systems

Deliverable: D4.1 Preliminary run-time models and support for energy, performance and other qualities

Due date of deliverable: 31-05-2019

Actual submission date: 31-05-2019

Start date of Project: 01 June 2018 Duration: 36 months

Responsible WP4: Tampere University (of Technology)

Revision: final version

Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Service	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (excluding the Commission Services)	

DOCUMENT INFO

Authors (alphabetical order)

Author	Company	E-mail
Francisco Barranco	UGR	fbarranco@ugr.es
Lubomír Bulej	CUNI	lubomir.bulej@mff.cuni.cz
Santiago Cáceres	ITI	scaceres@iti.es
Tiziana Fanni	UNICA	tiziana.fanni@diee.unica.it
Dip Goswami	TUE	d.goswami@tue.nl
Keijo Haataja	HURJA	keijo.haataja@hurja.fi
Pekka Jääskeläinen	TUT	pekka.jaaskelainen@tuni.fi
Jiří Kadlec	UTIA	kadlec@utia.cas.cz
Francesca Palumbo	UNISS	fpalumbo@uniss.it
Jukka Saarinen	NOKIA	jukka.saarinen@nokia.com
Raúl Santos de la Cámarra	HIB	rsantos@hi-iberia.es
Pablo Sánchez	UC	sanchez@teisa.unican.es
Carlo Sau	UNICA	carlo.sau@diee.unica.it
Shayan Tabatabaei Nikkhah	TUE	s.tabatabaei.nikkhah@tue.nl
Luis Medina Valdés	7SOLS	luis.medina@sevensols.com

Document history

Version	Date	Change
V1.0	31-05-2019	Final version for review.

Document data

Keywords	runtime platforms, runtime models, runtime adaptation

Editor Address data	Name:	Lubomír Bulej
	Partner:	CUNI
	Address:	Faculty of Mathematics and Physics Charles University 118 00 Prague Czech Republic
	Phone:	+420 95155 4189

Distribution list

Date	**Issue**	**E-mailer**
31-05-2019	Final	fitoptivis-all@lists.utu.fi
		Patrick.vandenberghe@ecsel.europe.eu

Table of Contents

1. Executive Summary

This report represents deliverable D4.1, one of the outcomes of Task 4.1 in WP4 of the FitOptiVis project. The main objective of WP4 is to deal with the complexity of application runtime management while considering a diverse set of heterogeneous platform components and configurations. The WP4 solutions provide instances of the WP2 reference architecture described in deliverable D2.1.

In the first iteration, this deliverable provides an overview of runtime platforms that will serve as platform components as defined in deliverable D2.1. These technologies span different levels of abstraction and serve to satisfy applications with diverse set of requirements. Specifically, we describe a latency-managed edge-cloud platform for latency sensitive cloud applications, a distributed OpenCL-centric heterogeneous device runtime software stack which provides a unifying backbone to applications relying on hardware accelerators, both local and remote, and the CompSOC platform for applications targeting execution on system-on-a-chip.

To enable adaptive control of application quality attributes (e.g., image resolution and quality, or frame rate) in response to resource availability and the desired quality trade-off, the runtime platforms need to provide means for resource managers to control application parameters linked to individual quality attributes and to manage resources assigned to an application. Each of the platforms enables adaptation at different levels of abstraction and at different time scales. To facilitate design of the necessary management interfaces, the deliverable also reports on adaptation scenarios relevant to use cases from various partners contributing to WP4.

The content of this deliverable contributes to MS3 (Preliminary components and methods release with standalone assessment).

2. Introduction

Work package 4 addresses Objective 3 of the FitOptiVis project:

> ***Objective 3**: Real-time multi-objective combinatorial optimisation; data and process distribution; run-time adaptation through virtualization; run-time quality and resource management; energy driven adaptations; workload (re-)distribution; support for run-time upgrades.*

Specifically, in WP4 the consortium develops techniques for run-time resource management within the system architecture template outlined in WP2. The main goal is to deal with the complexity of application runtime management, reconfiguration, and monitoring, while considering a diverse set of heterogeneous platform components and configurations. To increase developer productivity and to promote vendor independence with respect to compute platform, this diversity should become transparent from an application developer’s point of view. Task 4.1 focuses on run-time technologies and models to support management of performance, energy, and other qualities. This deliverable reports on the outcomes of this task in the first year of the project and outlines initial plans for the second year.

In Chapter 3, the report provides an overview of technologies and concrete platforms that will serve as a basis for virtual reconfigurable platforms as defined in the FitOptiVis reference architecture (see deliverable D2.1). To satisfy the diverse set of requirements found in FitOptiVis use cases, multiple concrete platforms are needed, each tailored to serve different types of requirements. For example, while real-time applications with modest latency requirements and a time frame for reconfiguration measured in seconds may be well served by a solution utilizing a general-purpose compute cluster in an edge-cloud environment, a hard real-time application implementing a tight control loop may need to utilize custom FPGA accelerators to meet latency requirements. Building a single unified hardware, software, and tooling framework to satisfy vastly different requirements would be neither possible, nor desirable. Instead, in FitOptiVis we aim to unify on the level of concepts, principles, and abstractions to find and extract commonalities found in different domains.

A common theme of FitOptiVis systems is the support for runtime management of various quality aspects through adaptive adjustment of configurable system parameters. Such a task is generally implemented using a MAPE-k loop [KEP03], which generalizes the concept of a control loop for adaptive systems. Such control loops can be nested to form a hierarchy of control loops operating at different time scales. This approach can be applied also in the FitOptiVis project, where a top-level MAPE-k loop can be thought to operate with the time frame of seconds, determining the setpoints for a lower-level control loop operating at the time scale of milliseconds or even microseconds. The presented technologies are intended for solutions operating at different time scales.

Section 3.1 describes a *multi-node managed-latency* private *edge-cloud* platform that will provide probabilistic guarantees to parts of applications (time-sensitive services) with *soft real-time* requirements that will be deployed in the edge-cloud. The platform is expected to support solutions with reconfiguration time frames in seconds, which can be either general soft real-time services, or top-level adaptation control loops managing set points for lower-level control loops. By focusing on probabilistic guarantees, we aim to reduce the impact on developers by not requiring them to express application performance requirements through many low-level metrics, but rather through a simple end-to-end metric on application probe points.

Servicing especially interactive applications targeting shorter time frames, Section 3.2 provides a description of a *distributed*, *heterogeneous-device* runtime software stack based on OpenCL, which can be used to spread the execution of application's computational tasks to all available resources (local or remote), and which can be fully controlled from the application running on a terminal device. The foundations for an extensible and portable heterogeneous system-software stack had been laid out in the ALMARVI project, and will be used and extended in FitOptiVis to support new use cases in distributed and reconfigurable computing. This part directly addresses the objective of managing the complexity of a heterogeneous distributed execution platform and allowing an application to harness all available resources through a standardized API. Because OpenCL can encapsulate all types of compute devices ranging from general-purpose CPUs to fixed-function accelerators, the consortium believes that the diversity management goal is well met by relying on it as a backbone, by enabling easy support/integration path for the various hardware-software platforms developed in the project by partners, and by extending the standard whenever needed. Section 3.2 also lists potential extensions to the OpenCL API identified during the first year of the project that will enable runtime monitoring, among other requirements associated with a distributed, dynamically changing environment. OpenCL supports defining heterogeneous task graphs via its command queue abstraction, which provide a basis for distributed heterogeneous task scheduling envisioned to be done later in the project, which also helps Task 3.2 (Programming and Parallelization Support).

A higher-level programming model, OpenMP 4/5, is being added as an example of an end-user programming language on top of the developed stack. This addresses the goal of *transparency*. Because the OpenMP view of the platform components is more restricted than that of OpenCL, more decisions on the suitable devices for each function are delegated to the management layers in the stack, instead of relying solely on the programmer in this choice. The developed OpenMP 4/5 offloading support on top of the OpenCL based stack is described in Section 3.3.

For the lowest-level solutions operating at the shortest time frames, Sections 3.4 and 3.5 discuss two hardware-software platforms that will be supported and extended in the project: The CompSOC platform for composable and analysable hard real-time applications running on a single system-on-a-chip, and platform templates tailored for Xilinx Zynq based FPGA SOCs as an easy-to-use implementation and prototyping platform. Both platforms target and support high-performance embedded computations, but place themselves in different layers of the work done within FitOptiVis: CompSOC defines a complete framework for design and implementation of hard real-time applications which utilize resource sharing, while the presented FPGA platforms enable prototyping and integrating of any hardware platforms with ease. The presented Xilinx-based platforms make a connection to the design flows in WP3 (Design-time support) allowing to prototype and utilize new hardware IP in combination with already commercialized ones running in the same system as described in WP5 (Devices and components).

Chapter 4 deals with support for adaptation in the runtime platforms and applications built on those platforms. To support different trade-offs between various quality aspects (visual quality, resolution, latency) and resource usage (compute resources, I/O bandwidth, memory consumption), the architectural description of FitOptiVis applications (see Deliverable D2.1) will enable binding individual quality aspects to corresponding resource requirements. It will also expose configurable parameters that allows a runtime entity, e.g., an adaptation manager, to request a particular quality level for a specific aspect. Such an adaptation manager will then control the individual parameters to achieve a higher-level goal, e.g., best overall quality given fixed amount of resources, minimal resource usage, best quality possible, etc. The adaptation manager needs to closely co-operate (or be integrated) with the platform runtime in order to ensure that the resource requirements associated with the desired levels of different quality aspects are satisfied.

Similarly as in Chapter 3, we have to deal with adaptation at different levels of abstraction corresponding to the supported runtime platforms. In Section 4.1 we, therefore, provide an overview of adaptation support in the context of the managed-latency edge-cloud platform, where the system needs to manage deployment of applications to individual nodes as well as allocation of resources such as CPU time, memory, and I/O bandwidth to co-located applications. For applications targeting systems-on-a-chip implementation and have shorter time frames, Section 4.1 presents an overview of adaptation mechanisms and management interfaces on the CompSOC platform, along with mapping of CompSOC concepts to the FitOptiVis reference architecture. Section 4.3 provides an overview of adaptation support for reconfigurable hardware, and Section 4.4 collects adaptation scenarios related to use cases from partners contributing to WP4, detailing their particular use case/example application and focusing on the type of runtime challenges that they face along with their current solution plans.

Chapter 5 provides a short conclusion and envisioned next steps for the second year of the project.

Finally, to provide a basis for new contributions in FitOptiVis, a survey of existing virtualization and resource management techniques is included in Appendix A.

3. Runtime Platforms

This chapter provides an overview of technologies and concrete platforms that will serve as a basis for virtual reconfigurable platforms as defined in the FitOptiVis reference architecture. We describe the platform model and the correspondence to architectural concepts defined in WP2 (Reference architecture, virtual platform and integration), i.e., the “instantiation” of the WP2 architecture on a specific platform. Each platform serves to satisfy a different subset of the diverse requirements present in FitOptiVis use cases and is intended to applications operating at different time frames.

3.1 Managed-Latency Edge-Cloud Environment

Modern Cyber-physical Systems (CPS) rely on data from sensors and perform computationally-intensive tasks on the data (computer vision, data analytics, optimization, and decision making, learning and predictions) which often cannot be executed on edge devices due to the limited energy budget and computational power.

To obtain the necessary computational power, such systems are typically split into parts that execute on edge devices and parts that execute in the cloud. However, the connection with the physical world inherent to CPS requires these systems to operate and respond in real-time, whereas the cloud was primarily built to provide average throughput through massive scaling. The real-time requirements impose bounds on response time, and when executing tasks in the cloud, a significant part of the end-to-end response time is due to communication latency.

The concept of edge-cloud aims to tackle this problem by moving computation to computational clusters that are physically closer to edge devices. While this reduces communication latencies, edge-cloud alone does not guarantee bounded end-to-end response time, which becomes more dominated by the computation time. The reason is that while the cloud itself focuses on optimizing the average performance and the cost of computation, it does not provide any guarantees on the upper bound of the computation time of individual requests. To satisfy the needs of modern cloud-connected CPS we need an approach that can reflect the real-time requirements of modern CPSs even with cloud in the computation loop.

3.1.1 Probabilistic Latency Guarantees

Strict latency guarantees on each individual request are the domain of real-time programming, which comes at a very high price, as it forces developers to use a low-level programming language, severely limits the choice of libraries, and imposes a relatively exotic programming model of periodic non-blocking real-time tasks.

We instead advocate the use of standard cloud technologies (i.e., micro-services running in a container-based cloud such as Kubernetes) and modern high-level programming languages (e.g., Java, Scala, Python). However, we restrict ourselves to a class of applications for which soft real-time guarantees are enough (i.e., the guarantee on the end-to-end response is probabilistic, such as “in 99% of cases the response comes in 100ms and in 95% of cases the response comes in 40ms”).

It turns out that this is acceptable to a wide class of applications including augmented reality, real-time planning and coordination, video and audio processing, etc. Generally speaking, this class comprises any application that has a safe state and has a local control loop that keeps the application in the safe state while computation is done in the cloud. Consequently, the soft real-time guarantee pertains to qualities such as availability and optimality, but not to safety. In the context of the FitOptiVis project, which generally focuses on developing distributed image and video processing pipelines, this applies to many of the use cases (augmented reality, habit tracking, municipal speed cameras, etc.).

3.1.2 Probes and Latency Requirements

One of the goals of our work is to minimize the impact of using a managed-latency edge-cloud environment on application developers. Given that we aim to use standard cloud technologies, we also envision the developer creating artefacts, e.g., for the Kubernetes (K8S) platform. The only required extension is the specification of application real-time requirements in the application deployment descriptor.

Contrary to common cloud deployment practices, we aim to spare the developer from dealing with the selection of VM type, the number of virtual CPUs, memory, IOPS, etc. Similarly, we aim to avoid specification of auto-scaling rules (including triggers), because we consider these to be implementation details of the cloud platform’s internal mechanisms which the developer is not equipped to set correctly without an experiment.

We instead work with an abstraction in which the developer is responsible for providing the application and its soft real-time requirements, while the responsibility for assessing the performance of the cloud application, as well as allocating resources (i.e., the required number of virtual CPUs, memory, IOPS, etc.) and making scheduling and deployment decisions so as to ensure that the (probabilistic) guarantees are met, lies with the cloud platform. Consequently, if the platform determines that it cannot satisfy the requirements, it will not admit the application for deployment.

Specifically, when developing an edge-cloud application, the developer has to describe the application in terms of an auto-scaling micro-service with added communication latency requirements. In the specific case of the Kubernetes cloud platform, we extend the Kubernetes application deployment descriptor to allow declaration of measurement *probes,* special functions provided by the developer which the system uses to assess the performance of the application in a particular deployment scenario.

An example deployment descriptor for a sample face-recognition application is shown in Listing 1. The timing requirements for the application state that the response of the application on the “recognize” probe should be below 100 milliseconds in 99% cases, and below 50 milliseconds in 95% cases.

```
kind: Deployment
metadata:
  name: recognizer-deployment
  labels:
    app: recognizer
spec: # microservice specification
  template:
  metadata:
    labels:
      app: recognizer
  spec:
    containers:
    - name: recog
      image: repo/recog
      ports:
      - containerPort: 7777
      probes: # probes
      - name: recognize
      timingRequirements:
      - name: recognize limit
        probe: recognize
        limits:
        - probability: 0.99
          time: 100 # Max. 100ms in 99% cases
        - probability: 0.95
          time: 50 # Max. 50ms in 95% cases
```

Listing 1. Application deployment descriptor with timing requirements

A probe (or a set of probes) has to capture the essential behaviour of the application so that when invoked by the cloud-edge platform, it will provide a representative sample of the application's performance in the current deployment configuration. Expressing the application timing requirements over developer-supplied probes simplifies the specification of the contract between the application and the cloud-edge platform, and allows it to treat the application as a black-box.

3.1.3 Platform Status

The development of the managed-latency edge-cloud platform is in progress. During the first year of the project, several design iterations have been made and work on prototype implementation has been started. Inter-module interfaces, application middleware, and module prototypes have been implemented.

Given the experimental nature and possibly involved installation and configuration of the prototype, we plan to make the platform available as a hosted service during the second year of the project. We will work closely with partners interested in deploying parts of their application in a managed-latency edge-cloud environment.

3.2 Heterogeneous Distributed Software Runtime Stack

The development of a single-node heterogeneous software stack based on OpenCL was initiated in the ALMARVI project. In FitOptiVis, this stack is being extended to support a distributed edge-cloud setup that can map the architecture models defined in WP2 to concrete run-time concepts of execution platforms and their topologies while supporting new devices developed with WP3 technologies and other devices and components of WP5.

The primary questions we seek answers in the runtime stack development for are:

- What are the workloads that need to be executed on local devices given 5G, WiFi6 and other high-speed low-latency wireless network technologies?
- Where are the latency bottlenecks when offloading interactive applications across such networks to cloud-edge servers?
- Can we distribute event synchronization to minimize communication due to back-and-forth synchronization between the “application device” and the cloud-edge servers?

These questions are approached by developing a proof-of-concept heterogeneous runtime that is optimized also for low-latency tasks and which can support also other types of computation offloading in addition to those based on frame serving (e.g. cloud gaming which has become popular in the recent years).

The software stack being developed is shown in Figure 1, while an example usage context is shown in Figure 2.

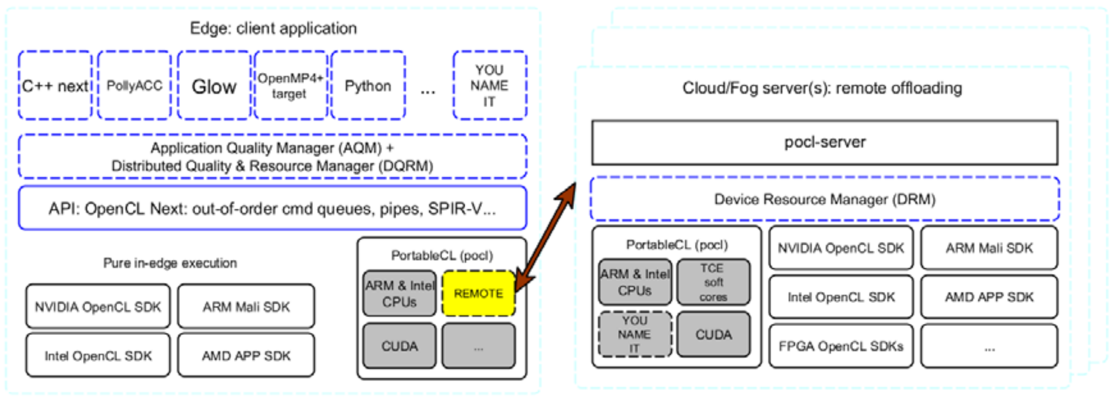


Figure 1. Multi-node heterogeneous distributed software runtime stack.

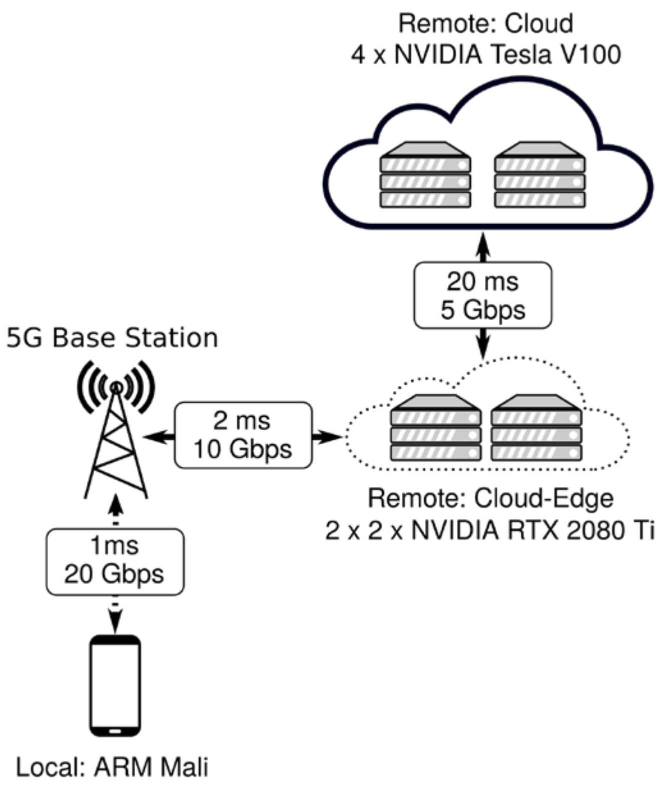


Figure 2. An example use context for the distributed runtime software stack. A terminal device (here a smartphone) deploys and starts the OpenCL application which then through a fast wireless link communicates to remote GPU devices in clusters at the cloud-edge and in the cloud.

3.2.1 OpenCL API Extension Candidates

The current notion is that OpenCL can serve as a good basis for a compute API both in local and distributed scenarios. However, already during the first year of the project, we identified the following features, which might be beneficial to add to the API (first as extensions and later as official part of the standard) to better support remote cloud-edge offloading scenarios:

- Platform: **Device Proximity**. The existing OpenCL API (practically) does not model connectivity between devices. Devices are assumed to reside in a single computer and to be accessible at most via a system bus such as PCIe or AXI, with shared external memory and/or per-device external memory. It would be beneficial to allow applications to make offloading decisions based on how efficiently devices are connected together: the API could be a platform-level query API with a possibility to query for the link between two devices. How the links are modelled and categorized is an open question at this point. E.g. 1) same shared memory hierarchy, 2) same system bus, 3) in the same local network, 4) internet connectivity
- Device: **Link Status**. Especially with 3) of the previous item and especially with 4), the performance of the link heavily depends on the simultaneous traffic and other varying conditions (e.g. the proximity of the nearest 5G base station). It would be useful to be able to monitor historical statistical information of the link's

performance in the past 5 seconds or e.g. the past 5 buffer transfers. Because it is hard to isolate the network part's time from the client side code, it might be useful as an OpenCL runtime API. Of course the most important link status information is whether the link is working in the first place, as it affects the reachability of the device.

- Device: **Reachability**. In OpenCL there is already a flag for 'availability' of the device. This might be reused for scenarios where a remote device is temporarily unavailable due to networking issues.
- Command Queues: **Performance History**. Auto-tuning scenarios attempt to execute a kernel on multiple devices while varying parameters that affect execution. While the information is natural to reside on the client side of the OpenCL API, it might be useful to provide some level of support in the runtime API for querying the estimated performance of the given kernel. The kernel performance estimate might be identified with a hash and input buffer sizes or similar. It might be difficult to design this API to fit OpenCL therefore it might be better to keep it in a client-side helper API layer.
- Command Queues: **Command's Energy Consumption**. Now the profiling command queues allow storing time stamps of events. In terms of tuning the power performance, it might be interesting to also record the consumed energy in case the target supports such information. This might be difficult to get accurate as it's hard to account for which kernel consumed the energy in the processor especially if there are multiple ones running. It's worth researching at least for the dedicated GPU farm scenario where we execute one kernel at a time and might then resort to average power numbers which can be multiplied with the execution time. The OpenCL API could be connected to the profiling command queues time stamping system: the time stamps could also record "energy stamps" at a similar incremental fashion.
- Command Queues: **More Profiling/Performance Counters**: Advanced profiling information could include the cache hit miss counter values in a similar stamping fashion with the same caveat as above: in case multiple kernels are executing at the same time, it might be difficult to isolate which kernel caused which part of the cache level misses.
- Device: **Temperature Readings** of the processor/memories or any other components equipped with a temperature sensor.
- Command Queues: **Real Time Commands with Execution Cancellation**: In some soft real time cases we can just reduce quality when a kernel takes too long time. It would be useful to provide mechanism to the command queues that allow killing a kernel when a time limit is reached. This could yield a special "timeout event" which other commands could listen to and kill also the next ones that are dependent on the regular finish event that the killed command should have produced.
- Buffers: **Unreliable Buffers**: This is connected to the soft real-time case and the cancelled kernels, and not delivering full data in time, but still producing some useful data. E.g. when we produce images in a tiled fashion, it may be useful to display a partially rendered/decompressed frame, especially when applying heavy filtering on top of it or when it's assumed that the incomplete frame in general looks OK if there are enough complete frames displayed per second.

- Buffers: **File-initialized Buffers**: Some of the buffer content could be initialized from files (possibly an URI) in the system where the remote Device resides. This is currently not possible in OpenCL as it only allows initialization from an array.

3.2.2 Distributed OpenCL Runtime Status

The distributed OpenCL runtime is being implemented within the Portable Computing Language (POCL) open source project, with internal releases made available to the project partners until the runtime becomes mature enough for general use by the open source community, at which point the code will be published at http://code.portablecl.org.

At the time of writing this document, the latest internal release available to project partners at https://github.com/cpc/pocl-fitoptivis was labelled as version 0.2 with the following feature highlights:

- Improved support for more complicated multiple-device setups
- Android build (see documentation for details)

To provide the reader with an idea on how remote offloading works with pocl-remote, brief usage instructions are given here, while a more detailed documentation, including build instructions, can be found at:

https://github.com/cpc/pocl-fitoptivis/blob/master/doc/sphinx/source/remote.rst

On the server, the `clinfo` command must list at least one OpenCL device. The server can be then started using the following command:

```
./server/pocld <IP ADDRESS> <PORT>
```

Note that `pocld` will listen on two ports, PORT and PORT+1. The amount of messages produced by the server can be adjusted by setting the `POCLD_LOGLEVEL` environment variable to the desired level before running `pocld`. The default log level is `err`. The server accepts the following log levels: `debug`, `info`, `warn`, `err`, `critical`, and `off`. On the client, the following environment variables need to be exported:

```
export POCL_DEVICES=remote
export POCL_REMOTE0_PARAMETERS=<IP ADDRESS>:<PORT>/<DEVICE ID>
```

The IP ADDRESS and PORT values are self-explanatory. PORT is the lower of the two port numbers assigned to the server. The DEVICE ID is the index of the device on the server. Valid indices range from 0 to N-1, where N is the total number of devices across all platforms on the server. The index is the order in which `pocld` lists the devices in the OpenCL platform it uses. This is the same order as displayed by `clinfo`.

The `clinfo` tool can be used to perform a "smoke test" to ensure that the distributed setup works. When configured properly, the tool should also list remote devices:

```
$ clinfo|grep pocl-remote
Device Version OpenCL 1.2 CUDA HSTR: pocl-remote 123.456.789.123:10998/0
```

A simple dot-product example can be then run by executing the example1 binary:

```
$ cd examples/example1
$ ./example1
(0.000000, 0.000000, 0.000000, 0.000000) . (0.000000, 0.000000, 0.000000, 0.000000) = 0.000000
(1.000000, 1.000000, 1.000000, 1.000000) . (1.000000, 1.000000, 1.000000, 1.000000) = 4.000000
(2.000000, 2.000000, 2.000000, 2.000000) . (2.000000, 2.000000, 2.000000, 2.000000) = 16.000000
(3.000000, 3.000000, 3.000000, 3.000000) . (3.000000, 3.000000, 3.000000, 3.000000) = 36.000000
OK
```

3.3 Extension of the OpenMP Runtime Infrastructure

Since the 90's, OpenMP has been a major standard for parallel programming of Symmetric Multi-Processing (SMP) architectures with shared memory. During the last years, new OpenMP versions have extended the specification to heterogeneous architectures. In fact, the last releases of popular compilers (such as gcc and clang) support the last specifications of OpenMP (versions 4.5 and 5.x) that include runtime code offloading to different devices such as NVIDIA GPUs, Intel Xeon-Phi co-processor and multi-core architectures.

The OpenMP offloading methodology differs from the commonly used approaches such as OpenCL. In OpenMP, the offloaded code is compiled for all possible devices during compilation. The specific binary executable code for all possible devices is integrated in a common executable, resulting in a "fat binary file". During execution, the OpenMP runtime library identifies the available devices and allows executing the device-specific offloaded code. Other approaches, for example OpenCL, rely on compiling code for a particular device at runtime.

The main advantage of the classical OpenMP approach is that all offloaded code has been compiled before execution, providing an important reduction of execution time and avoiding runtime compilation errors. Additionally, performance and quality estimations can be included in the device-specific code during compilation to facilitate efficient runtime execution. The main disadvantage of this approach is that all possible devices must be supported by the OpenMP compiler. Additionally, the compiler can generate a very big (fat) executable binary with the execution code for all possible devices. OpenCL simplifies the process with a runtime compilation technique that facilitates new device integration. However, OpenCL requires specific host "C" code as well as additional time for runtime compilation and compilation error management.

Additionally, there are devices, such as FPGAs, that cannot be efficiently programmed with these methodologies. In order to generate an efficient FPGA implementation, the source code usually requires synthesis-oriented modifications, code re-writing or even specific implementations in a hardware definition language (HDL). The FPGA synthesis process is more complex than a classical software compilation and it sometimes requires several iterations with different requirements. For this reason, it is difficult to integrate an efficient FPGA-oriented synthesis process with a standard software compilation methodology.

In FitOptiVis, the consortium is developing a new OpenMP offloading methodology that explores solutions for these limitations. The new approach is based on two main techniques: source-code offloading and dynamic offloaded code management. In the next section, we define the requirements for the new approach.

3.3.1 Requirements for the New OpenMP Offloading

The runtime implementation developed within the consortium to support the new methodology aims to meet several requirements that can be summarized as follows:

1. During compilation, the compiler should include in the executable files the code of the threads that could be allocated in different computation resources at runtime.
2. There should be a methodology that allows developing new thread implementations after compilation, but before application execution. The methodology allows extracting the thread code from the executable file and defines mechanisms for dynamic loading of the new implementations.
3. During execution, the runtime infrastructure should identify all the available thread implementations. The new implementations will be dynamically loaded.
4. The runtime infrastructure provides dynamic thread allocation during application execution.
5. The runtime library provides information about the available thread implementations and well as identified computing resources.
6. The computing resource information could optionally include performance data, such as memory size and clock frequency.
7. The device-specific implementation of a thread could optionally include performance data, such as memory requirements, execution time or power consumption.
8. During code execution, the runtime library provides a methodology to facilitate thread runtime monitoring.

3.3.2 The OpenMP Framework

In FitOptiVis, the consortium is extending the standard OpenMP methodology to meet the requirements of the previous section. The application is parallelized with OpenMP (versions 4.5 or 5.x). The compiler generates an executable file that includes the "standard" implementations (CPU cores and NVIDIA GPUs that support CUDA programming) as well as a new target: the thread source code and/or a compiler intermediate representation. After compilation, a tool extracts the thread code that is compiled in a different environment. To integrate these additional implementations in the application code, a dynamic library based methodology has been developed. During execution, the runtime environment identifies the available implementations and allows selecting the current thread implementation. The extended OpenMP framework is shown in Figure 3. The framework currently supports the activities shown in the solid-green boxes, which implement the dynamic thread-implementation management at runtime. The compiler and standard OpenMP runtime library modifications are under development and they will be presented in a next deliverable.

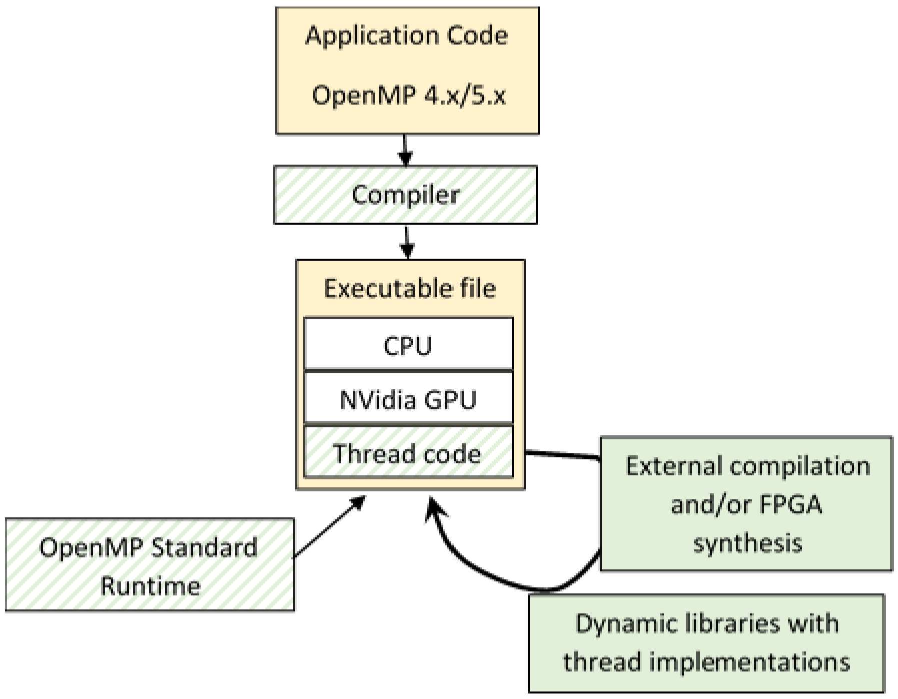


Figure 3. The Extended OpenMP framework.

3.3.3 OpenMP and OpenCL Integration

The methodology presented in the previous section has been extended to implement OpenMP threads in OpenCL to fit on top of the OpenCL-centric runtime stack described in Section 3.2. This integration allows supporting runtime compilation in OpenMP. From the thread code, an OpenCL kernel is generated. The current approach only transforms the C thread code of the OpenMP “parallel for” sections. Additionally, a library that provides support for OpenCL in OpenMP has been developed. This library synchronizes the OpenMP thread management and the OpenCL-based resource control. During execution, the application can select the OpenCL device that will execute the thread code. This code is compiled at runtime with the OpenCL API.

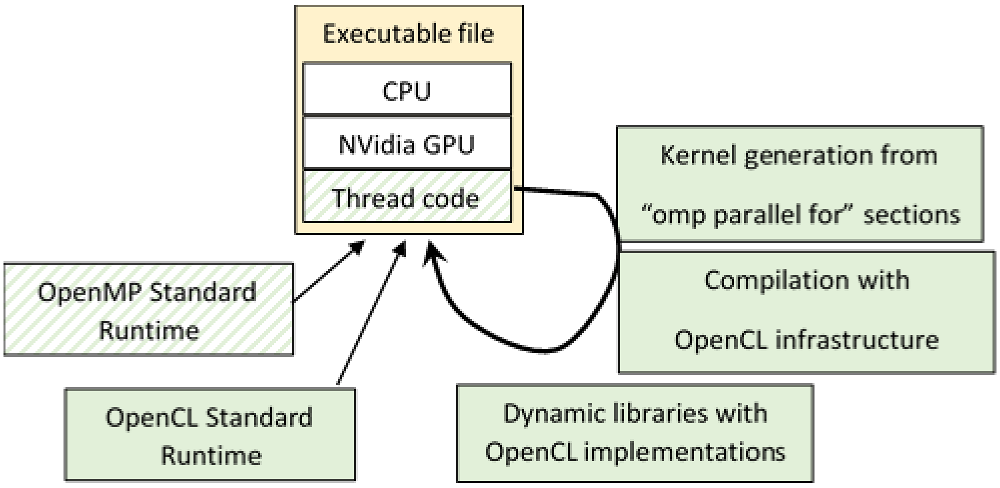

Executable file
CPU
NVidia GPU
Thread code
Kernel generation from "omp parallel for" sections
Compilation with OpenCL infrastructure
OpenMP Standard Runtime
OpenCL Standard Runtime
Dynamic libraries with OpenCL implementations

Figure 4. OpenCL integration in the OpenMP infrastructure.

This integration supports the Pocl-remote infrastructure presented in Section 3.2 and therefore it allows offloading OpenMP threads to remote devices.

3.3.4 OpenMP Extension Status

During the first year, the consortium has developed the methodology and the infrastructure to support dynamic thread implementations. Additionally, the integration of OpenCL in OpenMP has been demonstrated. During the next year, an open source compiler that supports OpenMP version 5.x (e.g. clang) will be modified to integrate the proposed methodology.

3.4 The CompSOC Platform

The CompSOC platform offers a Virtual Execution Platform (VEP) to each application. VEPs are entirely isolated from each other (space, e.g. memory, and time, e.g. TDM on processors or network-on-chip), such that each application can use its own Model of Computation and can be developed independently. This section is a summary of the platform description presented in [GOO17].

3.4.1 Hardware Architecture

MPSoCs contain multiple processors with local and shared memories. The processor's local memories are always on-chip Static Random-Access Memory (SRAM), close to the processor. Nonlocal memories shared between processors may be on-chip SRAM but often include off-chip Dynamic Random-Access Memory (DRAM). The latter has a much larger capacity (number of bits) than the on-chip memory, but at the cost of a longer execution time. Processors reach shared memories using a communication infrastructure, which is increasingly a NoC. A NoC is a miniature version of the Internet in the sense that communication is concurrent, is distributed, and is either packet based or circuit switched. As a result, it can run multiple applications of different criticalities at the same time. The CompSOC platform consists of multiple tiles interconnected by a

NoC. Tile types are master tiles, slave tiles, or a mix of both, and include processor tiles, memory tiles, peripheral tiles, etc.

3.4.2 Software Architecture

The CompSOC hardware platform contains computation, communication, and storage resources. Almost all can be shared between multiple requestors, and almost all can be (re)programmed at run time. The CompSOC software extends the single hardware platform to offer multiple Virtual Execution Platforms (VEPs). A VEP is an execution platform that is a subset of the CompSOC hardware platform, in terms of time (e.g., time multiplexing a processor) or space (e.g., non-shared DMA or a region in memory). Each application runs in its own VEP, which is created, loaded, started, and possibly stopped and deleted, at run time. A CompSOC platform can run multiple VEPs concurrently, without any interference between them, i.e., composably.

3.4.3 Microkernel and RTOS

Task arbitration can be classified along several axes. First, it may be absent when there is only one task on a resource. Otherwise it is required. Second, it may be preemptive or not. Third, arbitration may be static and follow a static-order schedule or be dynamic where the order of tasks is determined at run time. Multiple applications can share the processor using a microkernel such as CoMik, which arbitrates only between applications. Each application can use virtualized RTOS, such as µC-OS III, to independently arbitrate between application tasks.

An example CompSOC platform is shown in Figure 5 [GOO17].

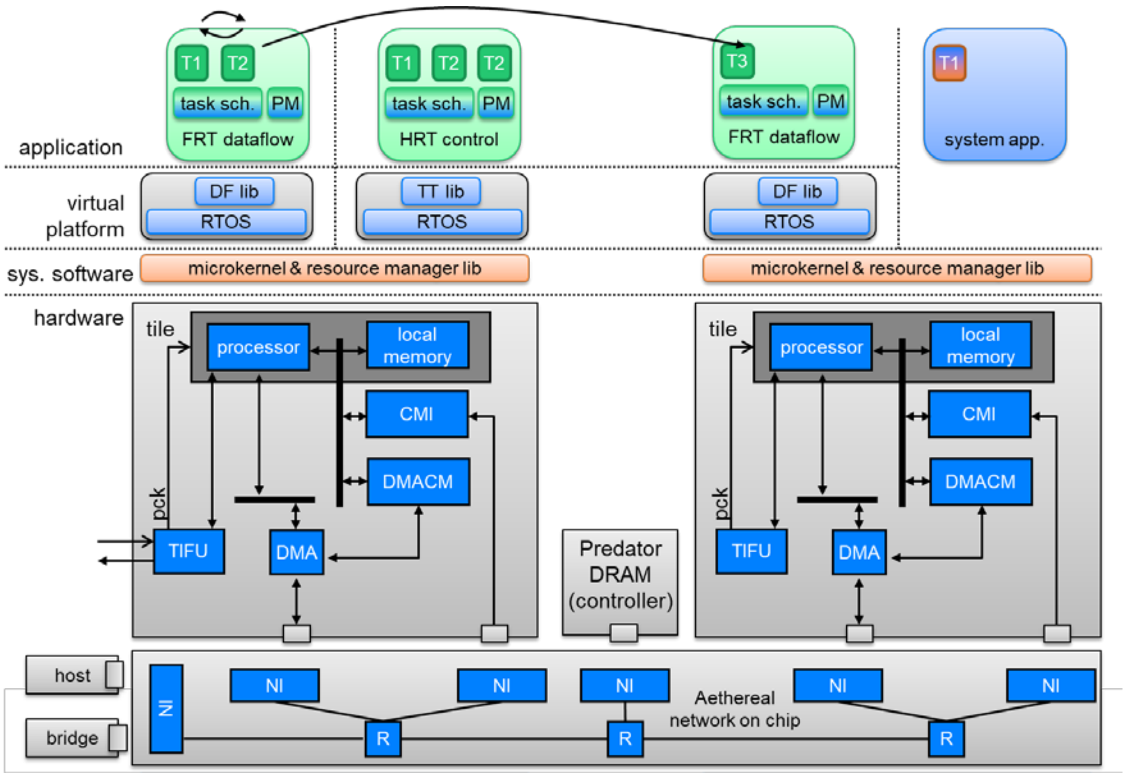


Figure 5. An example CompSOC platform.

3.5 Runtime for the Xilinx Zynq Platform

In contrast to the predecessor ALMARVI project, which only provided support for standalone boards and no board-to-board communication, the FitOptiVis project focuses on providing Peta Linux and Debian OS support, as well as enabling board-to-board communication in a local cloud.

The first version of design time and runtime support for the family of Xilinx Zynq and Zynq UltraScale+ systems has been developed by WP4 partners and released for use by project partners and general public by the end of April 2019. The new runtime provides support for Ethernet-based board-to-board communication in the local cloud, utilizing the Arrowhead framework, which is compatible with C/C++ clients running on ARM processors.

The following Xilinx Zynq systems are supported:

- **ZynqBerry (small)**. A small-size, low cost system with design time support developed in FitOptiVis. It has the Raspberry form factor and utilizes a 32bit Xilinx Zynq device (28nm) with small programmable logic area. WP4 provides support for Arrowhead-based board-to-board communication, Debian OS, and 32bit C/C++ clients. See [KAD18a], [TE0726], and [ARROW] for details.
- **Zynq UltraScale+ (medium)**. A medium-size system with design time support developed in FitOptiVis. Utilizes a 64bit Xilinx Zynq device (16nm) and reuses the carrier board and the Full HD video I/O FMC card from the ALMARVI project. WP4 provides support for Arrowhead-based board-to-board communication, 64bit Debian OS, and 64bit C/C++ clients. See [KAD18a], [KAD18b], [ARROW], [TE0820], and [TE0701] for details.
- **Zynq UltraScale+ (large)**. A large-size system with design time support developed in the FitOptiVis. The carrier board has the Mini-ITX form factor, utilizes a 64bit Xilinx Zynq device (16nm), and reuses the Full-HD video I/O FMC card from the ALMARVI project. WP4 provides support for Arrowhead-based board-to-board communication, 64bit Debian OS, and 64bit C/C++ clients. See [KAD18a], [KAD18c], [TE0820], [TE0808], and [TE080X] for details.

3.5.1 Inter-Cloud Connectivity with Arrowhead

The FitOptiVis (WP4) run-time resources are supported for the ZynqBerry board TE0726-03M by SW implementation of the Arrowhead framework compatible clients. The framework [ARROW] has been developed within the Artemis Arrowhead project and ECSEL Productive4.0 project.

In FitOptiVis WP4, we support the Arrowhead framework for board-to-board communication as a SW design time resource.

The targeted HW works with one Raspberry Pi3 board (bottom) and two ZynqBerry boards, as shown in Figure 6 below. The Raspberry Pi3 implements the Arrowhead framework [ARROW]. The ZynqBerry device on the top hosts a C++ producer capable of measuring the actual temperature of the Xilinx XC77010-1C device. The ZynqBerry device in the middle hosts a C++ consumer capable of requesting the temperature from the producer ZynqBerry board via the Arrowhead framework.

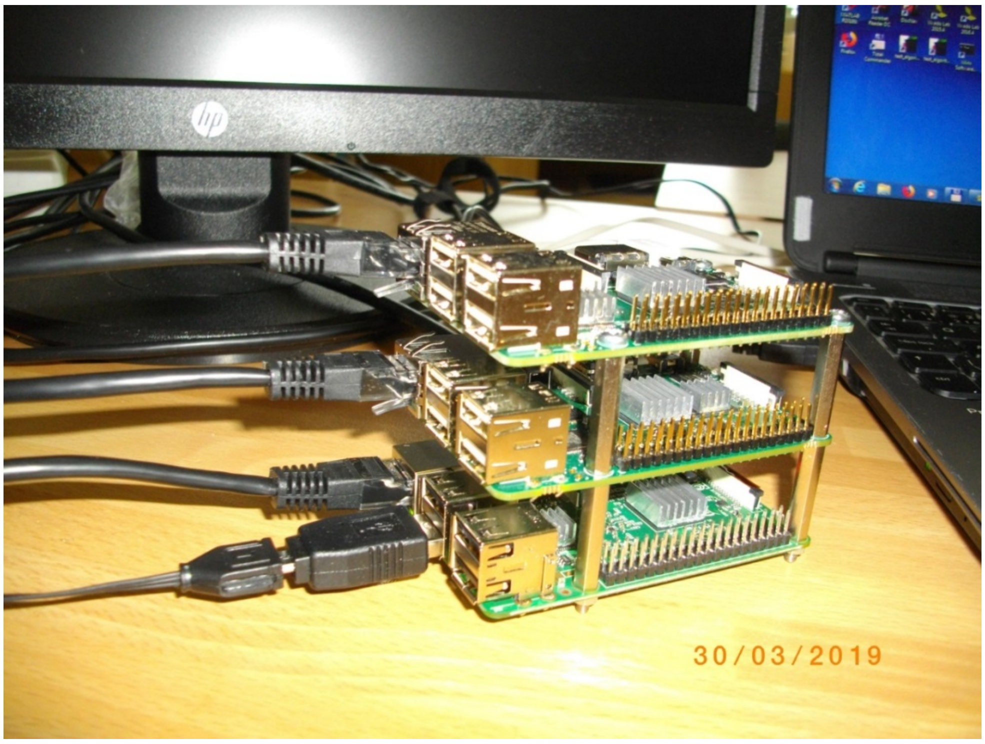


Figure 6. Raspberry Pi3, Arrowhead G4.0 clients on two ZynqBerry boards.

3.5.2 Obtaining Arrowhead Image for Raspberry Pi3

The client SW acts either as a Producer of a service, or as a Consumer requesting the service from the Arrowhead framework. The base hardware platform for the Zynq device is compiled with Xilinx Vivado 2018.2 tool.

To run and test Arrowhead clients, a light-weight implementation of the Arrowhead G4.0 framework needs to be installed and running on a Raspberry Pi3B board (RPi3).

Testing and running of the Arrowhead C++ clients on ZynqBerry boards requires Ethernet access to the Arrowhead framework services. It is recommended to use the precompiled image for the RPi3 board, which includes a pre-configured installation of the lightweight implementation of the Arrowhead G4.0 framework.

The image is available as one of the results of the running ECSEL JU project Productive4.0 at https://productive40.eu/, and is accessible to all Productive4.0 consortium partners. Please contact the coordinator of the consortium for further information regarding access to the light-weight implementation of the Arrowhead framework G4.0 running on the RPi3 board. After receiving the access to the download, unzip the three downloaded files Arrowhead-40-raspi.z01, Arrowhead-40-raspi.z02 and Arrowhead-40-raspi.zip into the final image file image_180626.img (size 3.711.959.040 bytes).

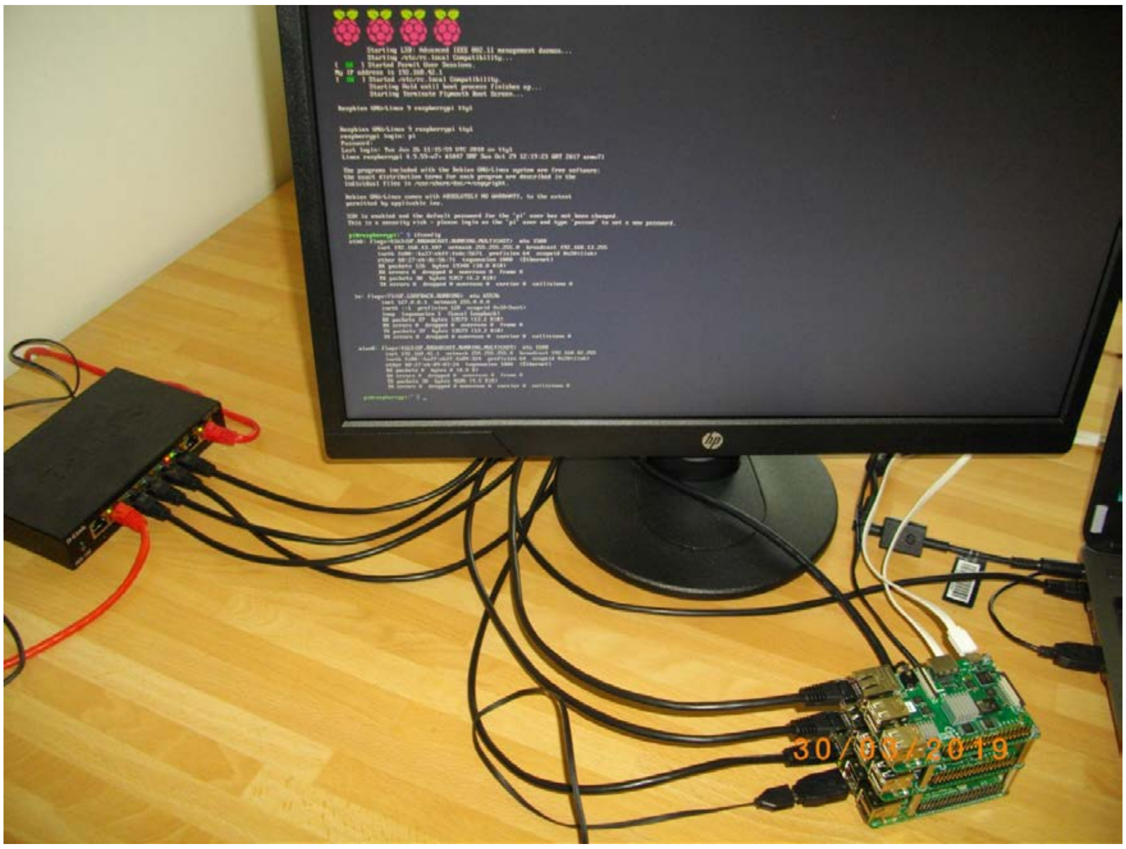


Figure 7. The Raspberry Pi3 will boot from the SD card, text output to monitor.

3.5.2.1 Installing Arrowhead Support on ZynqBerry Boards

At this stage, the Debian OS installed on the ZynqBerry boards can be updated to become compatible with the C++ demo applications implementing a service consumer and a service provider using the Arrowhead G4.0 framework.

- The installation of Raspberry Pi3 is described in Chapter 8 of App note [KAD18a].

3.5.2.2 Installing Arrowhead C++ Producer on Zynq Boards

The Arrowhead client SW acts as the Producer providing a service or as a Consumer requesting the service via the Arrowhead framework.

- Installation on ZynqBerry is described in Chapter 10 of [KAD18a].
- Installation on Zynq UltraScale+ is described in Chapter 11 of [KAD18b] and [KAD18c].

3.5.2.3 Installing Arrowhead C++ Consumer on Zynq Boards

The Arrowhead ConsumerExample can be compiled and run on the second ZynqBerry board. Alternatively, the ConsumerExample can be compiled and tested on the same ZynqBerry board as the ProviderExample.

- Installation on ZynqBerry is described in chapter 11 of [KAD18a].

- Installation on Zynq UltraScale+ is described in chapter 12 of [KAD18b] and [KAD18c].

3.5.3 Testing Arrowhead Operation on Zynq Boards

To test the client running on the Consumer Zynq board, first execute the ConsumerExample binary:

```
./ConsumerExample
```

The program should receive the following response from the ProviderExample program:

```
Provider Response:
{"e":[{"n": "this_is_the_sensor_id","v":26.0,"t": "1553675692"}],"bn":
"this_is_the_sensor_id","bu": "Celsius"}
```

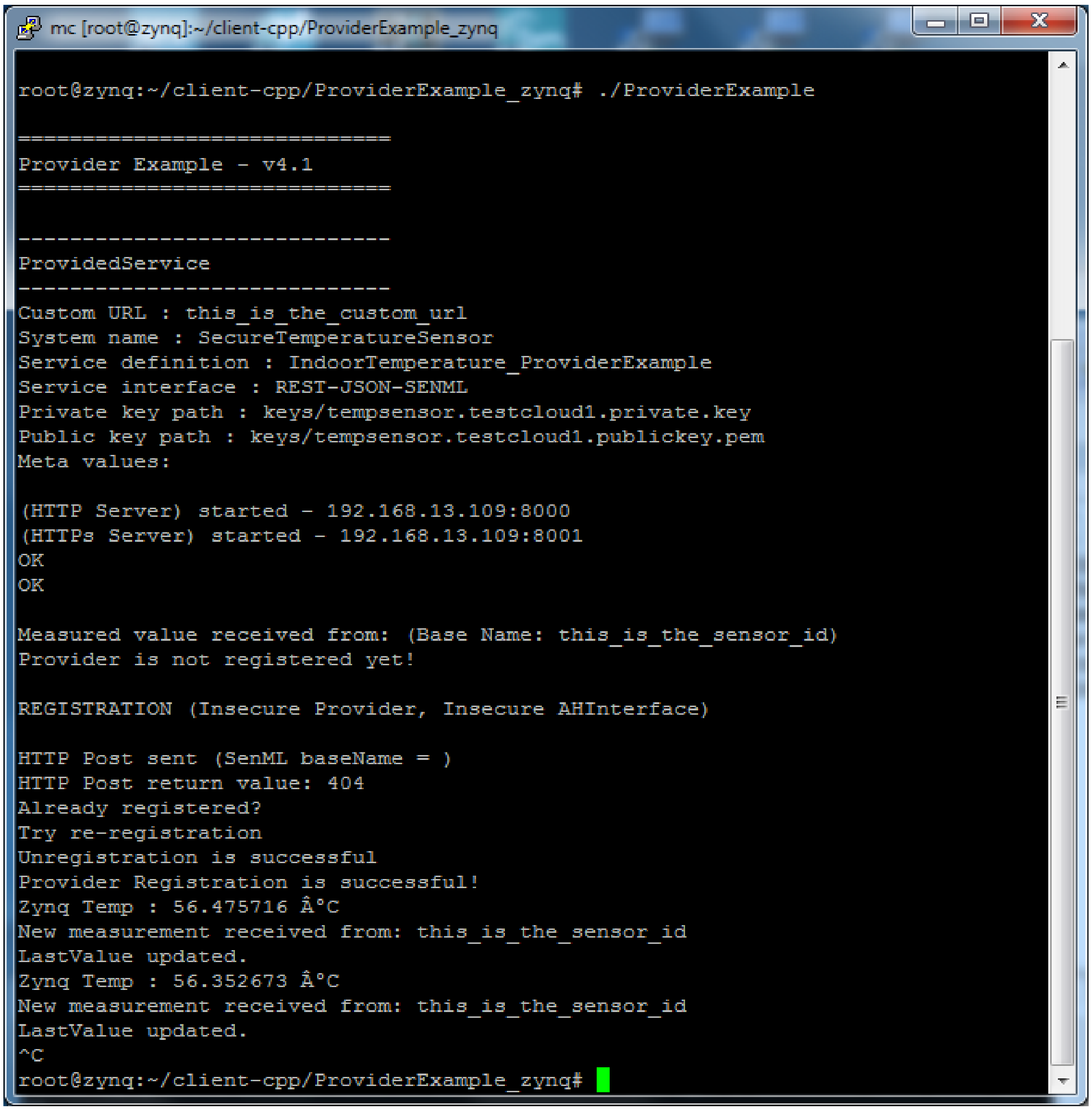
The first execution of the ConsumerExample program can fail if the Arrowhead database is used for the first time. The database of the Arrowhead framework running on the RPi3 has to be configured, and the ProviderExample and the ConsumerExample programs have to be connected by the database operator.

- The configuration of the Arrowhead database is described in Chapter 12 of [KAD18a].

Real temperature of the Xilinx chip on the Zynq board can be measured by a modified ProviderExample.cpp code, which is available in two variants, depending on the target platform:

- C++ code for the ZynqBerry platform is described in Chapter 14 of [KAD18a].
- C++ code for the Zynq UltraScale+ platform is described in Chapter 15 of [KAD18b] and [KAD18c].

All other files of the ProviderExample project remain identical. To test the modified provider on the Zynq boards, replace the code in the ProviderExample.cpp file and recompile the project using make.

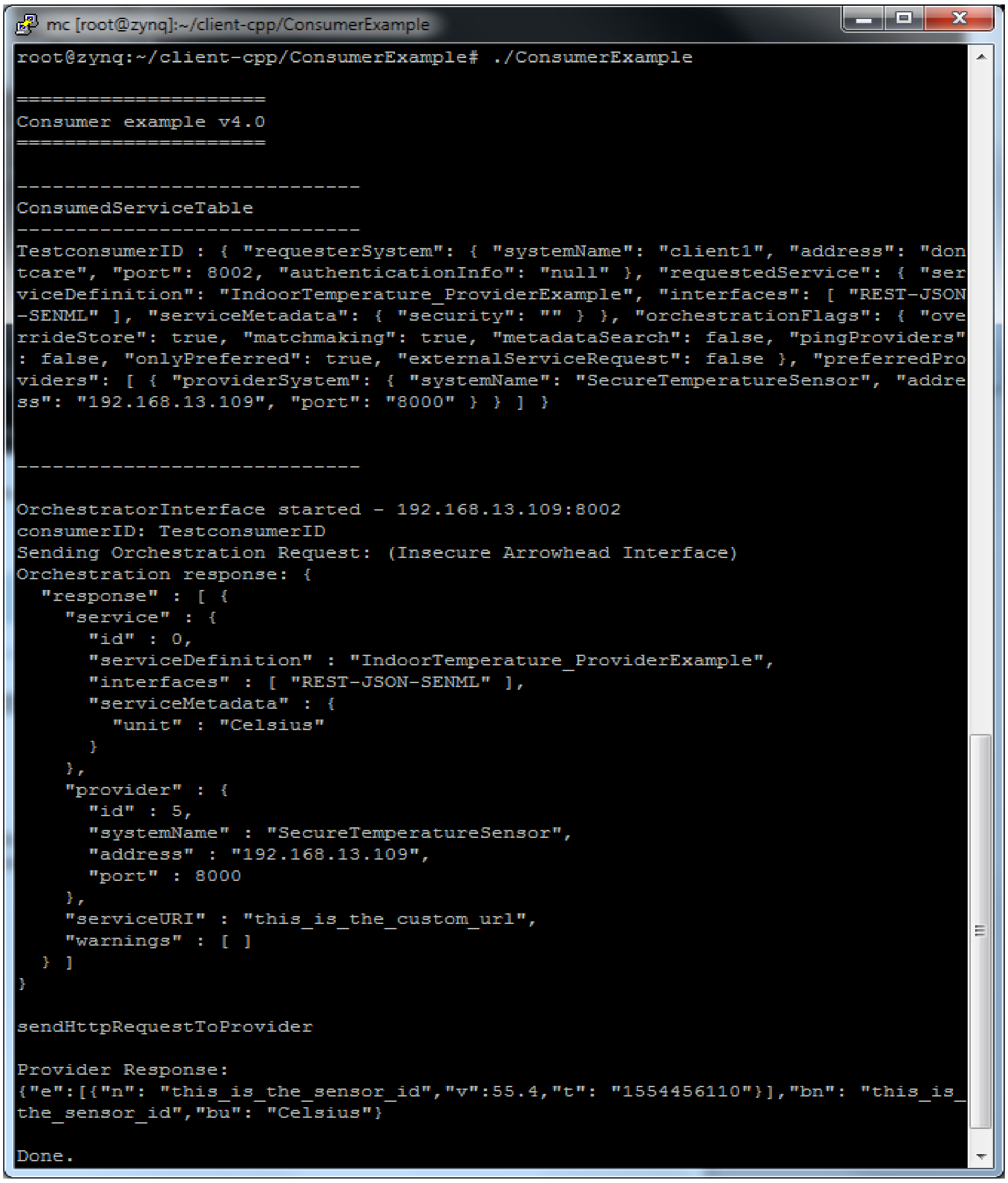


```
root@zynq:~/client-cpp/ProviderExample_zynq# ./ProviderExample

=============================
Provider Example - v4.1
=============================

-----------------------------
ProvidedService
-----------------------------
Custom URL : this_is_the_custom_url
System name : SecureTemperatureSensor
Service definition : IndoorTemperature_ProviderExample
Service interface : REST-JSON-SENML
Private key path : keys/tempsensor.testcloud1.private.key
Public key path : keys/tempsensor.testcloud1.publickey.pem
Meta values:

(HTTP Server) started - 192.168.13.109:8000
(HTTPs Server) started - 192.168.13.109:8001
OK
OK

Measured value received from: (Base Name: this_is_the_sensor_id)
Provider is not registered yet!

REGISTRATION (Insecure Provider, Insecure AHInterface)

HTTP Post sent (SenML baseName = )
HTTP Post return value: 404
Already registered?
Try re-registration
Unregistration is successful
Provider Registration is successful!
Zynq Temp : 56.475716 Â°C
New measurement received from: this_is_the_sensor_id
LastValue updated.
Zynq Temp : 56.352673 Â°C
New measurement received from: this_is_the_sensor_id
LastValue updated.
^C
root@zynq:~/client-cpp/ProviderExample_zynq#
```

Figure 8. Registration (re-registration) of Provider to Arrowhead framework.

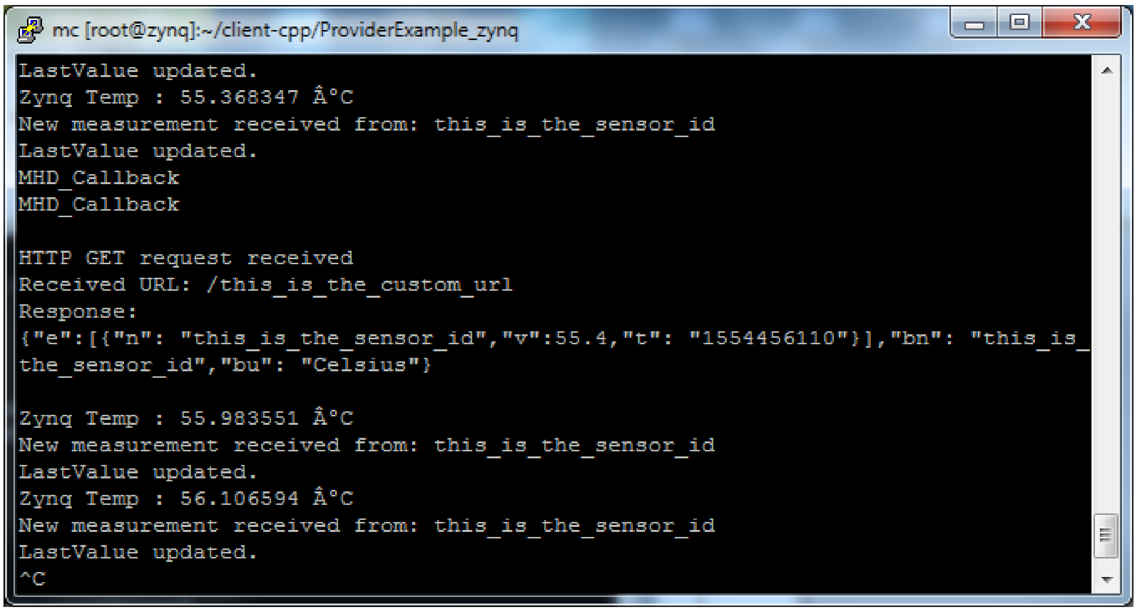
When executed, the modified ProviderExample is registered in the Arrowhead database. For debug purposes it also prints the actual temperature of the Zynq chip to console, as shown in Figure 8. The output of the Consumer requesting the actual temperature is shown in Figure 9, while Figure 10 shows the output of the Provider receiving and handling the request.

The measurement of the real temperature of the chip is an example of application-independent run-time information that may need to be collected for a system consisting of multiple boards to autonomously adapt computation to environmental conditions. Additional application-specific metrics, e.g., frame rate, may need to be collected to track performance of the system.



```
root@zynq:~/client-cpp/ConsumerExample# ./ConsumerExample

=====================
Consumer example v4.0
=====================

-----------------------------
ConsumedServiceTable
-----------------------------
TestconsumerID : { "requesterSystem": { "systemName": "client1", "address": "don
tcare", "port": 8002, "authenticationInfo": "null" }, "requestedService": { "ser
viceDefinition": "IndoorTemperature_ProviderExample", "interfaces": [ "REST-JSON
-SENML" ], "serviceMetadata": { "security": "" } }, "orchestrationFlags": { "ove
rrideStore": true, "matchmaking": true, "metadataSearch": false, "pingProviders"
: false, "onlyPreferred": true, "externalServiceRequest": false }, "preferredPro
viders": [ { "providerSystem": { "systemName": "SecureTemperatureSensor", "addre
ss": "192.168.13.109", "port": "8000" } } ] }


-----------------------------

OrchestratorInterface started - 192.168.13.109:8002
consumerID: TestconsumerID
Sending Orchestration Request: (Insecure Arrowhead Interface)
Orchestration response: {
  "response" : [ {
    "service" : {
      "id" : 0,
      "serviceDefinition" : "IndoorTemperature_ProviderExample",
      "interfaces" : [ "REST-JSON-SENML" ],
      "serviceMetadata" : {
        "unit" : "Celsius"
      }
    },
    "provider" : {
      "id" : 5,
      "systemName" : "SecureTemperatureSensor",
      "address" : "192.168.13.109",
      "port" : 8000
    },
    "serviceURI" : "this_is_the_custom_url",
    "warnings" : [ ]
  } ]
}

sendHttpRequestToProvider

Provider Response:
{"e":[{"n": "this_is_the_sensor_id","v":55.4,"t": "1554456110"}],"bn": "this_is_
the_sensor_id","bu": "Celsius"}

Done.
```

Figure 9. Consumer reads temperature of the Zynq chip via Arrowhead.



```
LastValue updated.
Zynq Temp : 55.368347 Â°C
New measurement received from: this_is_the_sensor_id
LastValue updated.
MHD_Callback
MHD_Callback

HTTP GET request received
Received URL: /this_is_the_custom_url
Response:
{"e":[{"n": "this_is_the_sensor_id","v":55.4,"t": "1554456110"}],"bn": "this_is_
the_sensor_id","bu": "Celsius"}

Zynq Temp : 55.983551 Â°C
New measurement received from: this_is_the_sensor_id
LastValue updated.
Zynq Temp : 56.106594 Â°C
New measurement received from: this_is_the_sensor_id
LastValue updated.
^C
```

Figure 10. Provider handling a request from the Consumer received via Arrowhead.

4. Runtime Adaptation

To manage trade-offs between different aspects of quality (e.g.,frame resolution, quality, rate or latency) and resource usage (e.g., CPU time, memory usage, I/O bandwidth, or energy), the runtime platforms need to be able to modify configurable parameters in response to desired quality set points and changing conditions.

In this chapter, we review the developed mechanisms for runtime reconfiguration and resource management, and introduce some of the algorithms and techniques envisioned to achieve the desired trade-offs between quality (performance) and resource usage for selected systems. The latter part of the chapter includes partner descriptions of runtime adaptation scenarios in use case specific applications and contexts to serve as scenario descriptions for guiding the development in the next years of the project.

4.1 Reconfiguration in Managed-Latency Edge-Cloud

At the highest level of abstraction, the managed-latency edge-cloud infrastructure implements a MAPE-K loop [KEP03] as shown in Figure 11 to ensure that the runtime guarantees are satisfied even in face of continuously changing conditions. Each of the phases of the control loop has a distinct responsibility:

- **Monitoring**. In general, the monitoring phase is responsible for keeping the internal model of the system up-to-date. In the context of the edge-cloud platform, the controller monitors the state of the K8S cloud (nodes, pods, and other entities such as services and deployments) as well as the state and performance of individual applications, e.g., how often .
- **Analysis**. The analysis phase is responsible for finding a deployment configuration (an assignment of application components to nodes in the cloud) that satisfies performance guarantees. A Constraint Satisfaction Problem (CSP) solver is used to find feasible solutions (in which timing requirements can be expected to hold), while the controller is responsible for evaluating the feasible solutions and choosing from among them.
- **Planning**. In the planning phase, the controller determines if the desired configuration differs from the actual configuration and if necessary, prepares a sequence of actions to bring the cloud to the desired state.
- **Execution**. In the execution phase, the controller makes actual changes to the cloud platform, following the plan of actions produced in the planning phase. In many cases, the actions can be executed in parallel, except when there are explicit precedence constraints among tasks.

The four phases execute simultaneously, sharing data through a central **knowledge** component. In its simplest form, the knowledge component can be represented by a single centralized database. However, it is entirely possible for the knowledge component to interface with several storage back-ends that can be used for different purposes. As an example, we can consider the data storage, analysis, and visualization platform developed in the context of Task 4.2.

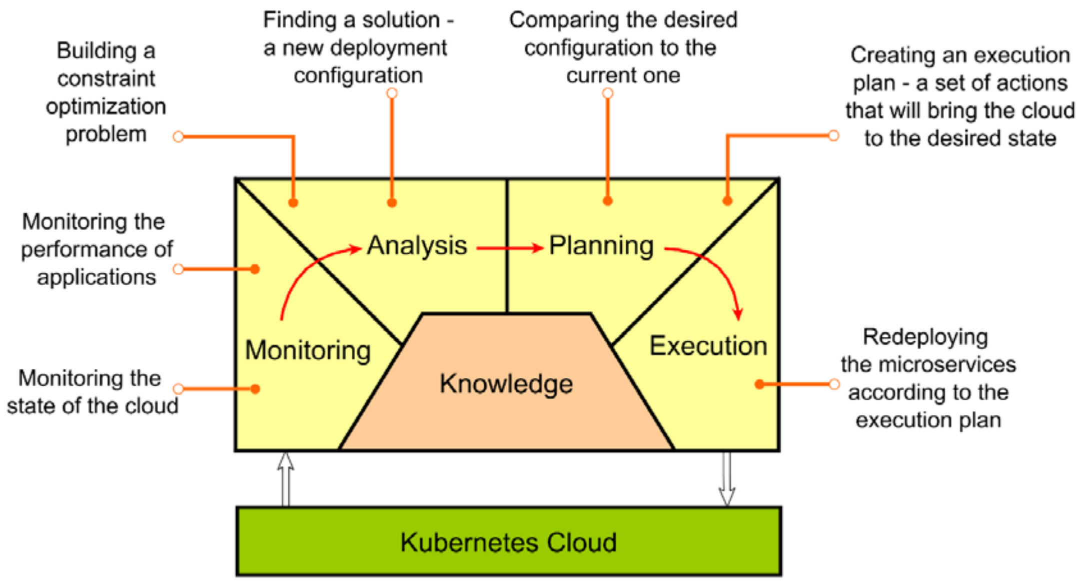

Building a constraint optimization problem
Finding a solution - a new deployment configuration
Comparing the desired configuration to the current one
Creating an execution plan - a set of actions that will bring the cloud to the desired state
Monitoring the performance of applications
Analysis
Planning
Monitoring
Knowledge
Execution
Monitoring the state of the cloud
Redeploying the microservices according to the execution plan
Kubernetes Cloud

Figure 11. Self-adaptation loop of the managed-latency edge-cloud platform.

Note that this control loop applies only to management of latency in the edge-cloud platform. FitOptiVis systems in the role of edge-cloud applications will implement application-specific higher-level (higher-latency) control loops responsible for configuring the set-points (e.g., resource limits, desired framerate) for a lower-level (low-latency) control loop responsible for achieving the desired set-points on the hardware components.

4.1.1 Edge-Cloud Platform Architecture

The architecture of the edge-cloud platform shown in Figure 12 comprises a number of modules, each with distinct responsibilities in the control loop. Yellow modules (need to) run on the master node, green modules do not (need to) run on the master node, and blue modules represent a middleware layer. We now elaborate on the role of individual modules and their interaction with other modules:

- **Event Cache**. The module is responsible for persistent storage of important events, such as changes in application deployment (requests to deploy or undeploy an application) and connections from unmanaged components. Unmanaged components execute outside the edge-cloud platform (e.g., a hardware accelerator) and connect (as clients) to the managed components executing in the cloud.
- **Knowledge**. Provides data storage and query capabilities to modules directly responsible for implementing the MAPE-K control loop. Knowledge data generally concerns cloud nodes (and their subtypes), application types and instances, and component types and instances.
- **Cloud Monitor**. Implements the monitoring phase of the MAPE-K control loop by periodically collecting information about the state of the nodes in the cloud, network latencies, and unmanaged components.
- **Analyzer**. Implements the analysis phase of the MAPE-K control loop and is responsible for finding an application deployment plan that satisfies the timing

requirements of all deployed applications. The module is internally split into Solver and Predictor submodules.

  - **Solver**. Responsible for finding the best deployment plan within a given time limit. Takes into account node utilization, network latencies, and predictions of component performance in deployment scenarios considered.
  - **Predictor**. Predicts performance of managed components, taking into account the hardware they are running on and the load induced by other components running on the same hardware.

- **Planner**. Implements the planning phase of the MAPE-K control loop, which means identifying differences between the current application deployment and the desired deployment. Constructs an ordered execution plan of tasks that need to be executed to transition the system to the next state.
- **Cloud Executor.** Implements the execution phase of the MAPE-K control loop by executing planned tasks either on the Kubernetes cloud, or on the other (Managed and Unmanaged) controllers.
- **Managed Controller**. Responsible for invoking probes on managed components and for reconnecting dependencies of managed component instances. Can access all Node Controllers at runtime.
- **Unmanaged Controller**. Responsible for reconnecting dependencies of unmanaged component instances from one managed instance to another, invoking probes on the client (which invoke managed components) to observe managed component performance including communication latency, and monitoring the state of unmanaged components.
- **Node Controller**. Runs on each node and monitors the utilization of a particular node and of all the components executing on that node (using standard K8S facilities for resource monitoring). In addition, it serves as a proxy to managed component instances for the Managed Controller.
- **Probe Controller**. Serves as a central entity through which all requests for probe invocation (on Managed and Unmanaged components) have to pass. Caches and forwards the results of probe invocations.
- **Network Controller**. Responsible for making changes in network configuration and for collecting network utilization data and connection latencies.

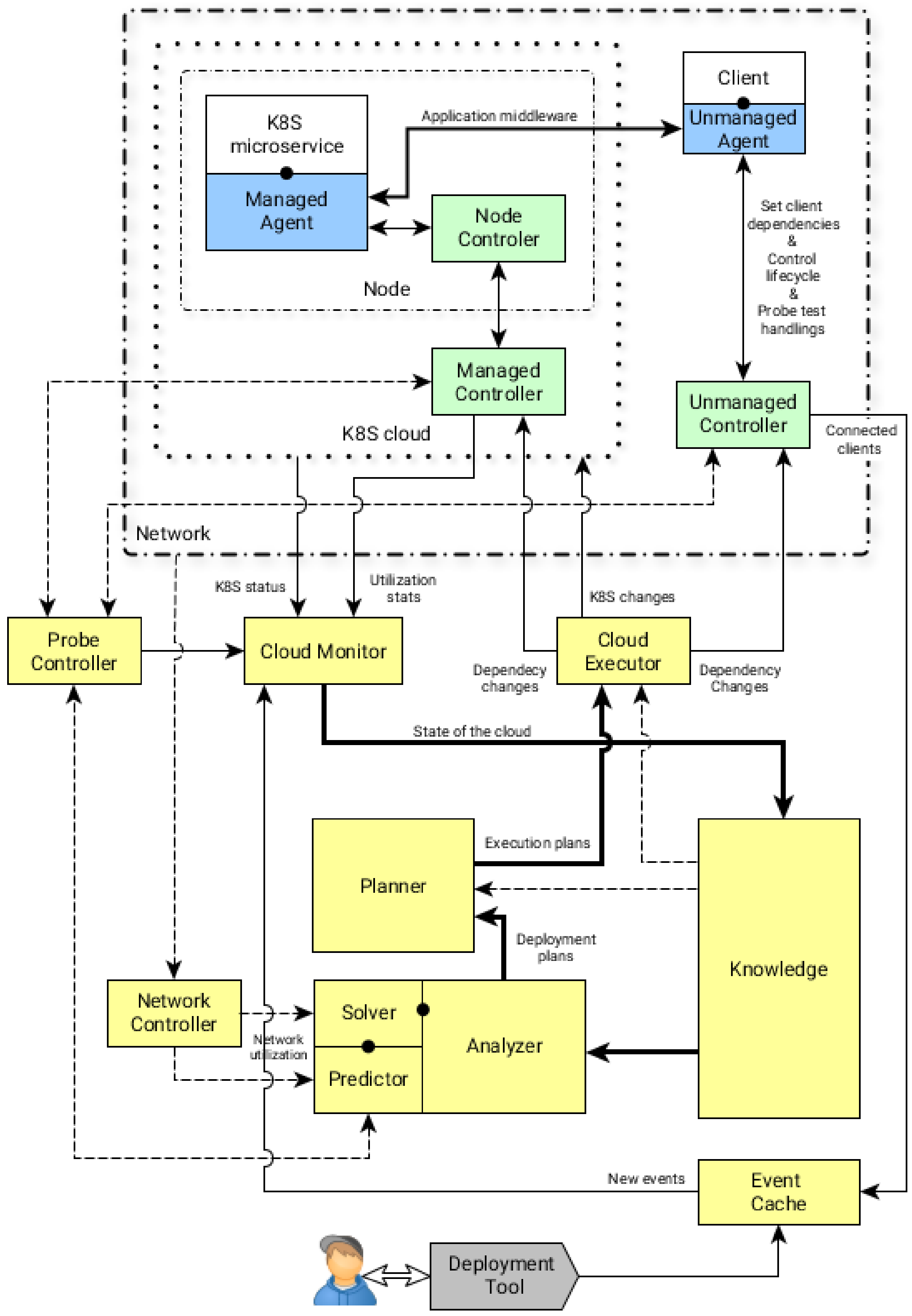


Figure 12. Architecture of the managed-latency edge-cloud platform.

4.1.2 Performance and Interference Models

To adaptively control deployment and redeployment of components in edge-cloud and thus to probabilistically guarantee end-to-end response time, the platform needs to build a model of application performance. This model needs to capture several modes of execution: baseline performance, when the application is exercised in isolation, performance under constrained resources, and performance in presence of other co-located applications sharing the physical hardware through virtualization.

Because we do not require the developer to provide the platform with apriori knowledge about application performance and resource requirements, the cloud platform needs to build the application performance model using experimental evaluation.

The model then is used to predict application performance in different situations, especially during admission control (when deploying a new application), and when optimizing the deployment of existing applications (to ensure that real-time guarantees are met, or to manage the utilization of cloud resources).

An important aspect of performance that the cloud platform needs to take into account is performance interference on shared resources (CPU caches, memory and IO bandwidth, etc.) when co-locating multiple virtual machines and/or containers on the same physical machine.

On the other hand, we generally consider the underlying network bandwidth unlimited for modelling purposes. The rationale behind this assumption is that edge-cloud applications are generally likely to be latency-sensitive, but not necessarily bandwidth-intensive – that would defeat the primary purpose of edge-cloud, which is to reduce communication latencies due to distance.

We also assume that edge-cloud infrastructure can generally be private, i.e., with significant level of control (like in hospital use cases). Consequently, we assume that the network infrastructure can be configured to assign time-critical network traffic a QoS class with high priority; that latency-sensitive services with guaranteed response time requirements will not saturate the network with bulk transfers; and that applications with excessive bandwidth requirements can be dealt with by proper network infrastructure design.

4.1.3 State of the Art

Cloud computing has been both a blessing and a curse. Cloud users can benefit from unprecedented availability and elasticity of resources, but the benefits come with strings attached. Cloud platforms have to continually balance the tension between efficient resource utilization (which determines costs) on the one hand, and quality-of-service guarantees demanded by latency-sensitive (LS) applications on the other hand.

Management of cloud resources has therefore become a vast and quickly moving research area, with many surveys mapping and categorizing the problems, challenges, and the state-of-the-art in various problem domains [CHE18, AMI17, HAM16, SIN15, FAN15, MAN15, GAR14]. In the context of our work we focus primarily on approaches to performance- and interference-aware self-adaptive systems which manage resource allocation and assignment in a cloud environment to achieve efficient utilization of available resources while allowing applications to meet their QoS target.

Q-Clouds [NAT10] is a QoS-aware control framework which transparently adjusts resource allocation to mitigate effects of interference on shared resources. Q-Cloud first profiles the virtual machines (VM) submitted by clients on a staging server to assess the amount of resources needed to attain the desired QoS without interference, and then manages the resources allocated to the deployed VMs in a closed control loop.

Cuanta [GOV11] is a technique for predicting performance degradation due to shard processor cache for any possible placement using a linear (as opposed to exponential) number of measurements. Applications are replaced by a synthetic clone which is tuned to mimic the application's cache pressure, and interference due to colocation is predicted based on a matrix of know interference effects between different configurations of cache clones. Even though Cuanta is not a full-fledged cloud scheduler, it was used to make better workload placement decisions for a given performance and resource constraints.

Bubble-Up [MAR11] avoids pairwise colocation profiling by characterizing the QoS degradation in LS applications using a synthetic workload with configurable memory subsystem stress test (the bubble), and the contentiousness of batch applications using a reporter workload with known sensitivity curve. The contentiousness of a batch application is mapped to a configuration of the bubble, which is then used to predict the interference inflicted by the batch application on the LS application.

Bubble-Flux [YAN13] improves on Bubble-Up by performing online profiling for LS workloads to account for workload phase changes and to identify more colocation opportunities.

Paragon [DEL13] is an online interference-aware scheduler, which uses collaborative filtering to classify incoming applications based on limited profiling signal and similarity to previously scheduled applications. It does not differentiate between batch and LS applications and schedules applications so as to minimize interference and maximize utilization. Applications are classified for interference tolerance using micro-benchmarks stressing a specific shared resource with tuneable intensity, which are run concurrently with an application to find out the interference level at which the application's performance falls below 95% of its performance in isolation.

Quasar [DEL14] improves on Paragon in that it also performs resource allocation instead of only resource assignment. Quasar extends the classification engine of Paragon to consider scale-out and scale-up scenarios, as well as different workload types with different constraints and resource allocation controls. It also provides an API that allows expressing the performance constraints regarding throughput and latency.

CloudScope [CHE15] is a representative of model-based approaches to QoS-aware cloud resource management and uses a discrete-time Markov Chain model to predict performance interference of co-located VMs. CloudScope runs within each host and collects application and VM-related metrics at runtime. The metrics serve to maintain an application-specific model capturing the proportion of the time an application uses a particular resource. The model is then used to predict slowdown due to colocation and ultimately to control placement of guest VM instances as well as adjusting the resources available to a hypervisor.

CtrlCloud [ADA17] is a performance-aware cloud resource manager and controller, which optimizes the allocation of CPU resources VMs to meet QoS targets. It maintains an online model of the relationship between allocated resource shares and the application performance, and uses a control loop to adapt the resource allocation so as to progress towards a probabilistic performance target expressed as a percentile of requests that must observe a response time within certain bounds.

Pythia [XU18] is a colocation manager which uses a linear regression model to predict combined contention on shared resources when co-locating multiple batch workloads with an LS workload. Pythia performs contention characterization for each batch workload running together with a particular LS workload and removes batch workloads that are too contentious to allow safe colocation. It then selects a small subset of batch workloads to co-locate with a latency sensitive workload and measures their combined contention to build a linear regression prediction model for contention due to multiple batch workloads.

Our selection illustrates a variety of approaches proposed over the years, each fitting a different context, yet none able to claim to solve the problem once and for all. Our approach will not be different in this aspect, but will focus on a privately-controlled cloud infrastructure. Unlike other approaches, we aim to treat all resources equally for the purpose of performance interference characterization, and rely on statistical characterization and similarity to reveal dependencies between background workloads.

4.2 Reconfiguration on the CompSOC Platform

The following presents the concept of reconfiguration and resource management framework to be realized on the CompSOC platform. This framework is an instance of the FitOptiVis architecture described in Deliverable 2.1. The section also describes how the concepts map to the abstractions provided by the OpenCL-centric runtime API.

4.2.1 Terminology

- **Component**: A component is a part of a platform or an application. Components can be composed to form larger components—e.g., applications or (virtual) execution platforms. They have one or more configurations, determined by component parameters, and may be reconfigurable. Component configurations have budgets and qualities. A budget can be provided or required. In OpenCL terminology, a *component* can be an OpenCL device (e.g. a GPU, CPU or an FPGA device) or an OpenCL *platform* (including all the controllable devices). It can also mean the whole OpenCL *application* including the host and the device parts, depending on the abstraction level used.
- **Task**: A task is an (application) component, which has only required budgets. In the OpenCL API, the kernels and buffer transfer *commands* are the tasks.
- **Application**: An application is a set of tasks that provides functionality to a user. In OpenCL the application consists of a main program running on a host device and a number of commands created by the program.
- **Resource**: A resource is a (platform) component, which has only provided budgets. This matches the concept of an OpenCL *device*.
- **Virtual Resource (VR)**: A virtual resource is a (platform) component, which is mapped to a single resource. In the case of pocl-remote, a virtual resource can be the device type/class/vendor for which an OpenCL kernel is optimized. Then the actual physical device will be assigned by the server-side resource manager.
- **Execution Platform (EP)**: An execution platform is the set of all resources. This matches the OpenCL *platform*.

- **Local Execution Platform (LEP)**: A LEP is the set of resources managed by a single Local Execution Platform Manager (LEPM). Every resource is part of a single LEP. Each compute server in the pocl-remote scheme can use a LEPM to manage its devices (e.g. which GPUs are dedicated to which remote application's use at which time).
- **Virtual Execution Platform (VEP)**: A VEP is a set of virtual resources that can host an application. An application has a valid deployment on a VEP when its required budgets match the budgets provided by the VEP.
- **Virtual Local Execution Platform (VLEP)**: A VLEP is a subset of a VEP that contains all VRs mapped to (resources that are part of) a single LEP. Each VLEP is managed by a Virtual Local Execution Platform Manager (VLEPM). The VEP/VLEP concepts currently do not have a direct counterpart in the OpenCL API, but these can be added within FitOptiVis as an additional initialization API by means of a runtime platform requirement description mechanism.

4.2.2 Overview

A block diagram of the proposed quality and resource management framework is depicted in Figure 13. Applications are composite components that are made up of tasks. Applications have one or more configurations, which are determined by application parameters. Applications may have certain provided qualities, and during their execution, they may be expected to provide certain quality levels (i.e., meeting QoS requirements). Each application configuration results in certain quality levels.

As shown in Figure 13, an Execution Platform (EP) is used to execute applications. In order to use the EP efficiently, applications are consolidated in an isolated manner. Subsequently, to realize this isolated consolidation, applications are deployed on Virtual Execution Platforms (VEPs). VEPs are composite platform components, which are comprised of virtual resources each of which must be mapped to a resource located in the EP. An application has a valid deployment on a VEP when its required budgets match the budgets provided by the VEP.

Applications may have certain quality requirements, which are met when they are properly configured and provided with sufficient resource budgets. Consequently, we propose a quality and resource management framework, which configures applications according to their quality requirements and ensures that application budget requirements are met. The proposed framework consists of several function blocks and databases, also shown in Figure 13. In the following section, we elaborate on the responsibilities of each block.

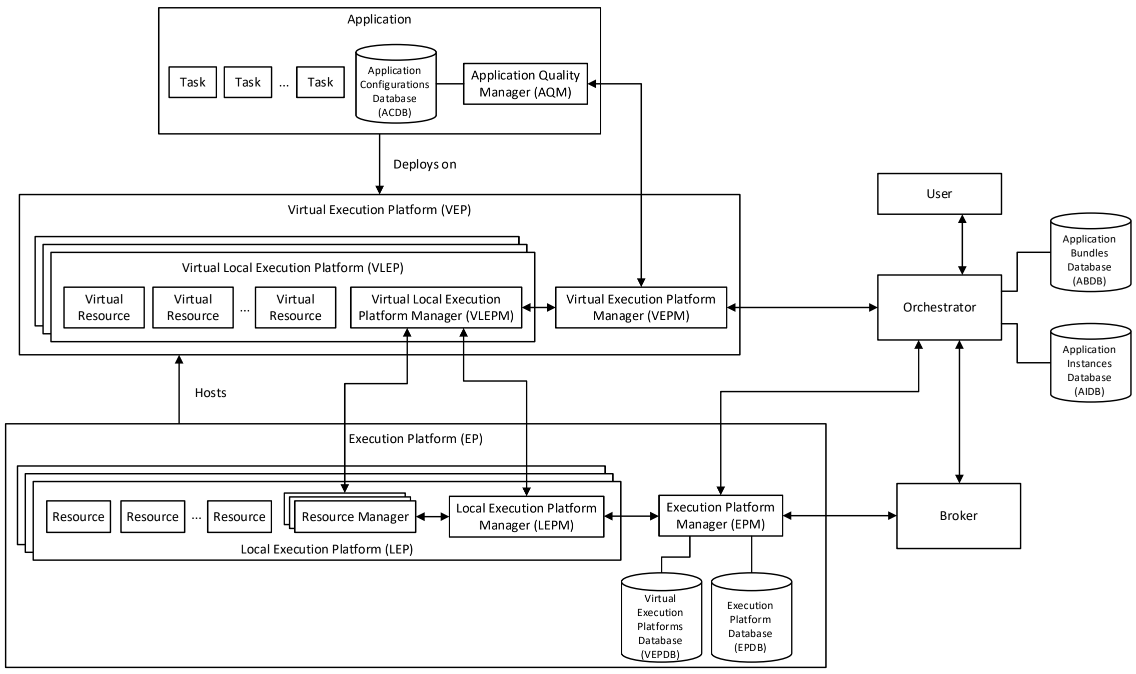

Application
Task
Task
...
Task
Application Configurations Database (ACDB)
Application Quality Manager (AQM)
Deploys on
Virtual Execution Platform (VEP)
Virtual Local Execution Platform (VLEP)
Virtual Resource
Virtual Resource
...
Virtual Resource
Virtual Local Execution Platform Manager (VLEPM)
Virtual Execution Platform Manager (VEPM)
User
Orchestrator
Application Bundles Database (ABDB)
Application Instances Database (AIDB)
Hosts
Execution Platform (EP)
Resource
Resource
...
Resource
Resource Manager
Local Execution Platform Manager (LEPM)
Execution Platform Manager (EPM)
Broker
Local Execution Platform (LEP)
Virtual Execution Platforms Database (VEPDB)
Execution Platform Database (EPDB)

Figure 13. Block diagram of the proposed quality and resource management framework.

4.2.3 Functional Blocks

4.2.3.1 Application Quality Manager (AQM)

The Application Quality Manager is responsible for lifecycle management of an application. Each application may have one AQM task, which performs application-specific functions such as configuring application tasks with proper parameters. In particular, it has the following responsibilities:

- Configuration and reconfiguration of applications during the instantiation and reconfiguration phases, respectively. Each application task may have certain parameters that must be set before the task starts to execute. Additionally, it may be necessary to modify these parameters during task reconfiguration. The AQM configures/reconfigures the application tasks using the parameters that are given by the VEPM.
- Measuring application qualities during application execution. A quality is a measurable value that demonstrates how effectively an application is operating. Each application may have certain quality requirements that must be met during application execution. Employing an application-specific method, the AQM measures and monitors application qualities at run-time.
- Making reconfiguration decisions when certain events happen. During application execution, certain events such as workload transitions may occur which necessitate application reconfiguration including modifying application allocated resources, application parameters, and/or application state (e.g., application tasks). Such reconfiguration decisions are made by the AQM.
- Sending reconfiguration requests to VEPMs. Since the AQM is not privileged enough to modify the application VEP, it must ask VEPMs to perform reconfiguration when the application VEP must be modified.

4.2.3.2 Orchestrator

The orchestrator, which serves as the entry point of the system, manages the execution of applications (i.e., instantiation and reconfiguration) by orchestrating the EPM and VEPMs. The orchestrator is responsible for the following:

- Receiving user requests regarding running and lifecycle management of applications. As mentioned above, the orchestrator is the entry point of the system. The end user sends its requests regarding loading (i.e., running) and lifecycle management (e.g., updating quality requirements) of applications to this entity.
- Management of Application Bundles Database (ABDB) and Application Instances Database (AIDB).
- Lifecycle management of Virtual Execution Platform Managers (VEPMs). Each application VEP is managed by a VEPM, and a VEPM itself is managed by the orchestrator. VEPM lifecycle management tasks such as VEPM instantiation are performed by the orchestrator.
- Management of application deployment. To deploy an application, the Orchestrator asks the Broker to select one of the application configurations and determine a VEP to host it.

4.2.3.3 Virtual Execution Platform Manager (VEPM)

The Virtual Execution Platform Manager is responsible for the lifecycle management of the VEP an application is deployed on. This is done through orchestration of VLEPMs. For each application, there exists one and only one VEPM. Upon user requests to instantiate an application, a VLEP is created, and the VEPM is loaded onto it by the Orchestrator. Subsequently, the VEPM creates VLEPs for VLEPMs, and manages the creation of application VEP by orchestrating the VLEPMs. The VEPM has the following responsibilities:

- Lifecycle management of VLEPMs. Each application VEP is distributed among several VLEPs, each managed by a VLEPM. VLEPM lifecycle management tasks such as VLEPM instantiation are performed by the VEPM.
- Lifecycle management of application VEPs. Lifecycle operations (including creating, destroying, and reconfiguration) of application VEPs are managed by the VEPM. Since an application VEP is composed of one or more VLEPs each of which managed by a VLEPM, its lifecycle management requires the orchestration of VLEPMs, which is performed by the VEPM.

4.2.3.4 Virtual Local Execution Platform Manager (VLEPM)

The Virtual Local Execution Platform Manager is responsible for the lifecycle management of a VLEP, which is a part of an application VEP. VLEPMs are instantiated by VEPMs and are responsible for lifecycle operations of VLEPs including creating, destroying, and reconfiguration of VLEPs. To do so, each VLEPM communicates with the LEPM and Resource Managers of the LEP it is mapped on. Constrained by its access rights, a VLEPM must ask the LEPM to reserve/release virtual resources. However, for other lifecycle operations, such as allocation and initialization, it directly asks the Resource Managers.

4.2.3.5 Execution Platform Manager (EPM)

The Execution Platform Manager is responsible for managing the resources that the Execution Platform (EP) is comprised of. All the global resource-related requests are passed to this entity. Additionally, it keeps track of available resources, their costs, and resources used by VEPs. In particular, the EPM is responsible for:

- Management of Execution Platform Database (EPDB) and Virtual Execution Platforms Database (VEPDB). The information regarding available resources, resource costs, and the resource shares owned by VEPs are collected and managed by the EPM in two databases. These information are provided by LEPMs.
- Exposing resource information to the Broker. During the resource brokering process, the Broker provides the EPM with a set of application required budgets and the maximum affordable costs. Having the global view of available resources, the EPM provides the Broker with a set of VEPs meeting the required budgets and costs.

4.2.3.6 Local Execution Platform Manager (LEPM)

As mentioned before, each resource is part of a LEP and is managed by a single Local Execution Platform Manager. LEPMs are entry points of LEPs. Resource-related requests sent by remote functional blocks are received by this entity. LEPMs are responsible for:

- Management of resource reservations and allocations. In order to create VLEPs, their required resources must be reserved and allocated. The actual reservations and allocations are performed by Resource Managers. However, given the fact that each VLEP may be composed of various resources, a single entity is necessary to ensure that all the required reservations and allocations are done successfully.
- Exposing resource information to the EPM. In order to keep the global view of EP updated, each LEPM informs the EPM about the available resources and their costs.

4.2.3.7 Resource Manager (RM)

Resource managers are employed to create, configure/reconfigure, and destroy virtual resources. As shown in Figure 14, several steps must be taken for each operation. To create a virtual resource, first, its required budget – described in the Budget Descriptor – must be reserved. In this step, the required budget is being compared to the budget provided by the resource. If the reservation is successful (i.e., the provided budget is not less than the required one), a virtual resource identifier is generated, and the creation process continues with allocating the resource. During this step, the budget is programmed into the resource using the identifier. Hence, the allocation step may take more time than the reservation step. After the allocation step, the virtual resource is created and it is ready to be initialized (i.e., to be configured, e.g., load instruction memory of a vCPU with application code). Finally, the initialized virtual resource starts running.

Similarly, several steps must be taken to destroy a virtual resource. First, the virtual resource must be stopped. Given the fact that the virtual resource may be busy at this point, stopping a virtual resource can be a slow process. After the resource becomes stopped, it may need to be reset to its initial state. Finally, the programmed budget must be released. When the budget is released, the available budget gets back to its previous state, and the virtual resource is destroyed. Besides lifecycle management of virtual

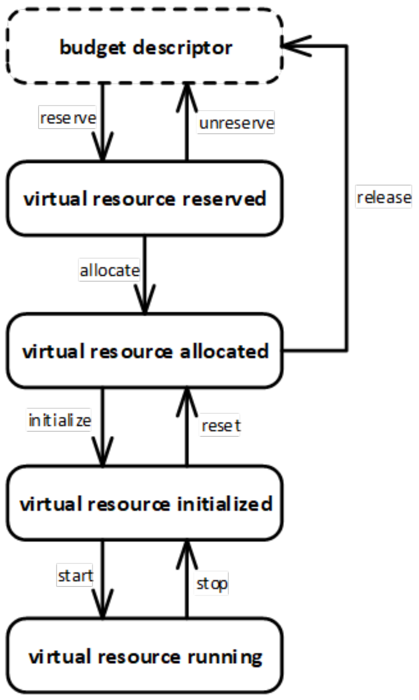


Figure 14. Lifecycle FSM of a virtual resource.

resources, RMs measure and monitor performance and costs of resources. In order to keep the LEPM updated about the status of local resources, RMs provide the LEPM with the measured performance and costs. Such provided data are maintained in the EPDB by the EPM.

4.2.3.8 Broker

The Broker, which acts as a decision maker in the system, determines the optimal configurations for all the platform and application components. For instance, when an application is planned to be instantiated, the Broker decides which application configuration should be deployed to meet the application's quality demands and which VEP configuration should be selected to host the application instance. To do so, the Broker needs to know information concerning application configurations (including their required budgets and offered qualities) and VEP configurations (including their provided budgets and costs). The former is provided by the Orchestrator using Application Bundles stored in ABDB, and the latter is provided by the EPM using the information stored in EPDB. The decisions are made in such a way that the application quality requirements are met and the aggregate cost of resources is minimized.

4.2.3.9 Databases

As shown in Figure 13, there are several databases in the proposed architecture containing information necessary for quality and resource management. Generally, the information of each component is stored in a structure called Component Bundle, shown in Figure 15. For each component configuration, the Component Bundle contains its parameters, qualities, Budget Descriptor, and initial state. Configurations are determined using the parameters. Qualities describe offered qualities of application components or costs of platform components. The Budget Descriptor, which has a hierarchical structure, describes either the provided budget of a platform component or the required budget of an application component.

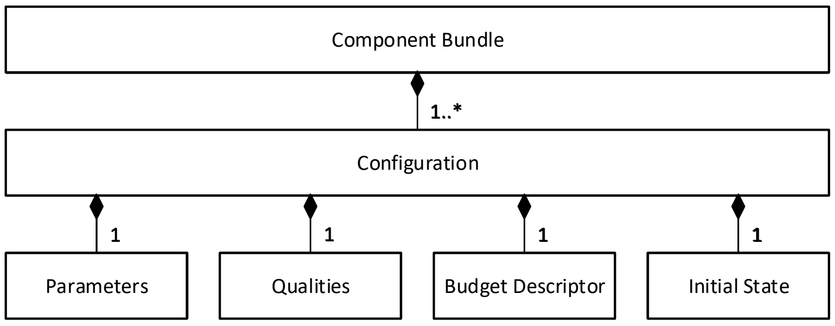


Figure 15. Structure of Component Bundle

The mentioned databases store the following information:

- Application Bundles Database (ABDB): This database stores all the application bundles. Each application bundle contains all the application configurations. This database is created and maintained by the Orchestrator.
- Application Instances Database (AIDB): It stores the bundles of application instances. Since each application instance is configured with one application configuration, the application instance bundle contains only one configuration. This database is also created and managed by the Orchestrator.
- Application Configurations Database (ACDB): The AQM needs to know about all the application configurations for making reconfiguration decisions. This database provides the AQM with this information. In essence, it stores the application bundle, which is also stored in the ABDB.
- Execution Platform Database (EPDB): It contains information of all the resources within the Execution Platform. This database is maintained by the EPM using the information collected from LEPMs.
- Virtual Execution Platforms Database (VEPDB): This database maintains information of all the created VEPs. Since each VEP is configured according to a single configuration, its bundle has only one configuration. This database is also maintained by the EPM using the information collected from LEPMs and VEPMs

4.3 Reconfiguration in Processor/Co-processor Systems

In general, designers should be supported at design-time, to define, characterize and be able to deploy platforms that optimally match the given requirements, while guaranteeing that customized applications are still interoperable. Nevertheless, in dynamic and reactive systems, such as CPS, design-time customizability is not sufficient.

Modern systems are required to be flexible and versatile, capable of supporting multiple operational profiles corresponding to different trade-offs, and capable of dynamic reconfiguration to switch between the profiles at runtime [BYS10]. We are therefore addressing the definition of efficient run-time methodologies capable of coping with these flexibility needs at all levels of CPS systems, from edge to cloud. Here we deal specifically with run-time adaptability at the hardware component level. More specifically, we refer to multi-purpose co-processing units.

The concept of dynamic parameter adjustment was introduced by Burleson et al. in [BUR01]. As illustrated in Figure 16, tuning processing in response to content variation and/or changing user/system requirements is made possible by runtime variation of different parameters. These can be classified as follows:

- **Functional parameters**. This kind of parameters allows tuning the output of a computation. They may include, e.g., filter and transform lengths, or quantization levels.
- **Architectural parameters**. This kind of parameters allows tuning guaranteed performance and energy consumption without modifying the output of the computation. They may include, e.g., the level of parallelism employed in the computation, which may affect throughput and energy consumption.

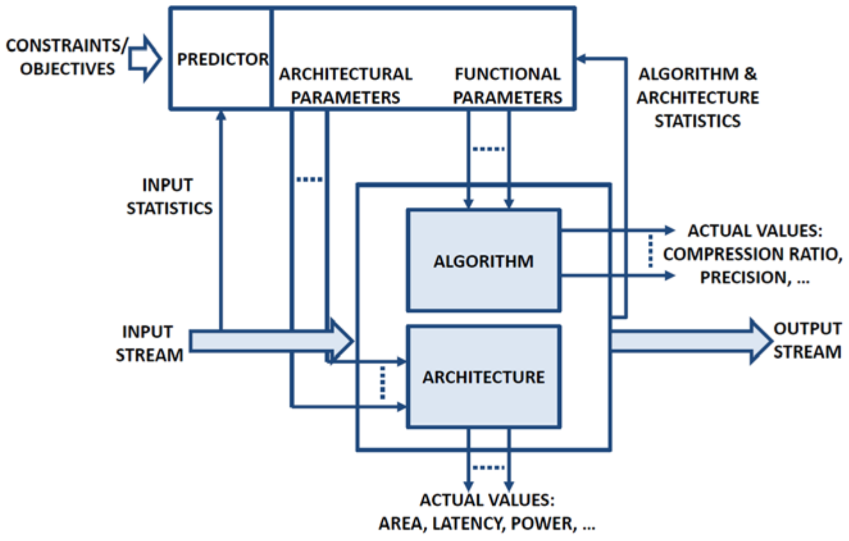


Figure 16. Dynamic parameter adjustment [BUR01].

The work of Burleson et al. refers mainly to video codec specifications, but it can be generalized to image and video processing pipelines such as the ones we are dealing with in FitOptiVis. In particular, this way of describing a dynamically tuneable computing infrastructure fits the work carried out in WP2 of the project, where a composable, customizable and reconfigurable virtual reference platform for video and image processing pipelines is defined. According to this formalism, both functional and architectural parameters can be customized to optimize a system before deployment to meet the given constraints, and to adapt the system at runtime to adjust to variable environmental or system conditions or to human requests.

The contribution specific to WP4 is at the predictor level. In processor to co-processor systems (see Deliverable 5.1 for more details on this kind of co-processing units) deployed using the Multi-Dataflow Composer (MDC, see Deliverable 3.1 for more details) coarse-grained functional and non-functional reconfiguration is enabled. In particular, MDC generated co-processors/accelerators are specialized hardware modules capable of accelerating different algorithms (functional reconfiguration) and/or different variants of the same algorithm (non-functional reconfiguration). Being applied at a coarse-grain, reconfiguration is very quick and takes place by simply overwriting a unique configuration register on the accelerator. Decisions on parameter tuning can be then taken at run-time, starting with the knowledge of the current state and taking into consideration varying objectives/requirements, characteristics of the processed data, and actual processing and architectural Key Performance Indicators (i.e. offered quality of service, throughput, energy consumption etc.).

Current status of this activity can be summarized as follows:

- We have completed the definition of the automated support for dynamic reconfiguration. The MDC tool is capable of automatically generating the APIs for transparent accessing of co-processors/accelerators from a host-processor and supports different types of coupling (e.g., loose coupling, utilizing memory-mapped communication, or tight coupling, utilizing stream-based communication), different host processors, and optionally using DMA for data transfers (WP3 & WP5 work). These APIs also enable reconfiguration of the co-processor/accelerator by simply changing the specific function call used to offload computation on the co-processor/accelerator.
- There are currently ongoing activities with respect to the definition of a proper, minimally invasive, monitoring infrastructure. Preliminary studies on the usage of HW performance monitoring counters are ongoing and we are discussing with other partners the best way to integrate our studies on HW monitoring, so as to implement the most efficient solution. Complete reporting and achievements on this activity are expected to be included in D4.3.
- Planned activities and preliminary discussions. These concern the predictor component, which is necessarily use case specific – the target platform and the use case goals determine specific details of this component. Starting from the low level requirements of the Water Supply use-case which are expected to be defined at M12, we intend to define use-case relevant run-time models. These models, taking as input the information coming from the above-mentioned monitors, are meant to be used within the predictor to report on system status and to enable (partially) autonomous decisions on runtime adaptation. Complete reporting and achievements on this activity are expected to be included in D4.2.

4.4 Specific Adaptation Scenarios

The following subsections provide details on adaptation specific to selected use cases.

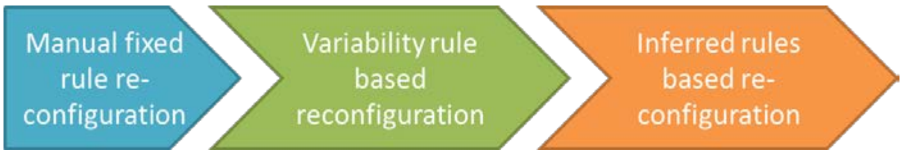
4.4.1 Modelling System Variants and Configuration Changes

In the design phase of computer vision systems, it is most often left to the human designer to model and define the rules that can be applied dynamically at run-time to achieve adaptability (e.g., choosing a reduced frame rate or picture size). This can be theoretically made quite effective but often requires manual fine tuning of these rules.

In FitOptiVis we aim for an incremental advancement over the current state of practice in this field. We rely on principles of reconfiguration and variability modelling coming from past efforts such as CVL [FLE09] and [LOP13]. In these modelling frameworks, variability is used as a means to construct Software Product Lines (SPLs). Such software product lines can be defined as a set of software intensive systems sharing a common, managed set of features that satisfies specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. The variability model makes explicit definitions of the constraints and requirements that define these software assets and how they relate to one another. Then, this information is used to make a change in the produced software via a set of different mechanisms (e.g., reconfiguration, code generation) at a given time in the process (design, time, compile time, link time or run-time), what is called in the literature the variability binding time [EID12].

In the course of the work in FitOptiVis WP4 we propose a mechanism to implement this in computer vision modules built for the UC3 Habit Tracking which control the distribution of load between edge nodes (running on computing-power constrained ARM devices) and more powerful cloud nodes (running similar algorithms but on powerful x86 nodes). The computer vision will be based on Deep Learning efforts such as OpenPose [CAO17] running on top of pyTorch1.

The general outline of the work to be undertaken in FitOptiVis is as follows:


Manual fixed rule re-configuration
Variability rule based reconfiguration
Inferred rules based re-configuration

We will start with a basic reconfigurable system relying solely on static rules provided by the designer (e.g., different configuration options triggered by rules defined by the developer). In second iteration, we will extend the system to leverage the variability models (based on BVR Tool [Vas15], used also in the ECSEL AMASS2 project) while still using rulesets provided at design time by human experts. In the final iteration, the system will be extended to use both rules provided by experts, but also rules inferred by AI systems using optimization techniques for the criteria of interest for the use case (e.g., energy consumption).

As of the writing of this document, work has started with experiments in the set-up of the UC-3 use case (Habit Tracking). The diagram in Figure 17 below illustrates the architecture of the system.

The system currently consists of a processing node in the cloud (running DenseCap deep learning network that uses the pyTorch framework on an x86 Linux machine) and a simulated edge node also using an x86 machine. The edge node will be migrated to Android/ARM-A in the next iteration. The system uses a single high level configuration

[1] https://github.com/pytorch/pytorch (PyTorch Neural Networks on Python)
[2] https://www.amass-ecsel.eu/content/bvr-tool-amass (Use of BVR in the AMASS project)

file (config.json in the image) which is provided to both systems and then locally interpreted using a static rule system (based on JSR-94 compliant rules). This transformation, which is currently static and based on hand-written rules will be changed into a more advanced system that uses variability for decision making as described in deliverable D3.1.

The high-level configuration is currently only interpreted once—during launch. Future iterations will include a reconfiguration system that can adapt to the environment, especially in the node (e.g., for low-energy scenarios).

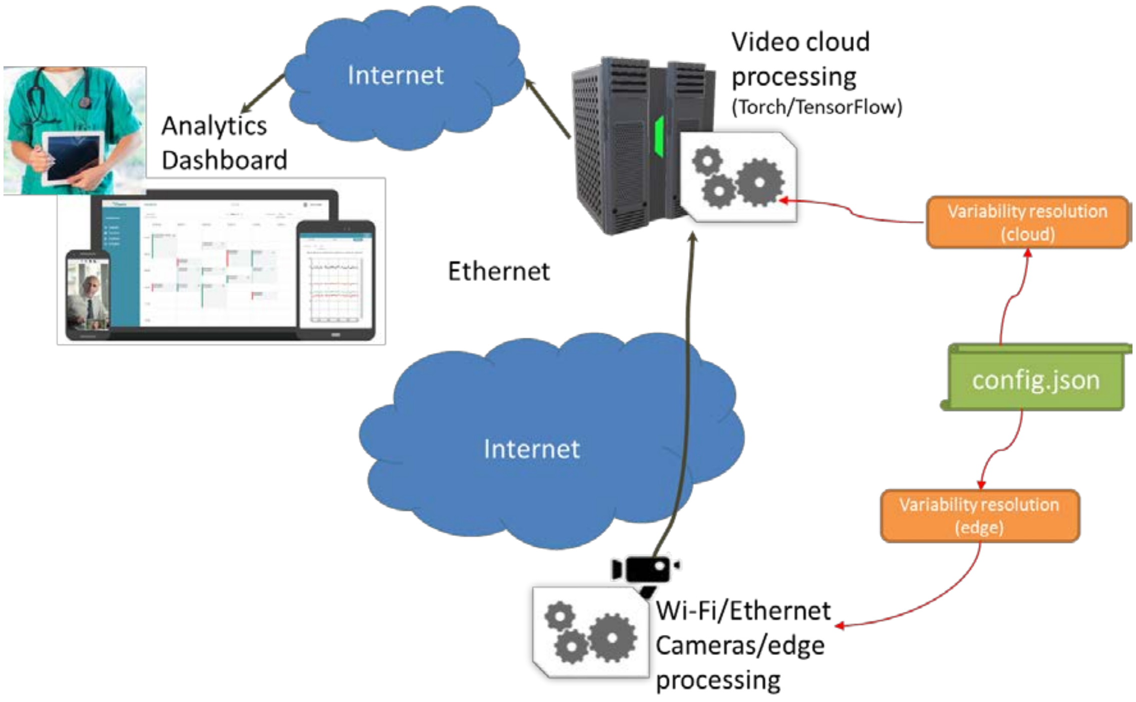

Internet
Video cloud processing (Torch/TensorFlow)
Analytics Dashboard
Variability resolution (cloud)
Ethernet
config.json
Internet
Variability resolution (edge)
Wi-Fi/Ethernet Cameras/edge processing

Figure 17. Architecture of the Habit Tracking system.

4.4.2 Selection and Compression of Task-Specific Features

During the first year in FitOptiVis, a study has been performed that assesses the feasibility of using visual attention to reduce data bandwidth in computer vision models for tracking and activity recognition applications. We have also considered other active vision approaches that include changing geometrical parameters of the sensor according to the task, or changing the perspective or focal length of the selected sensor. From current centralized methods, different metrics have been studied to be considered for the resource management (runtime or not), adopting the edge-cloud paradigm.

Some issues related with the streaming of video are: scalability when using multiple image/video sources, data bandwidth of the shared network, real-time performance of the video processing components, or privacy issues related with the transmission of images or the additional computational complexity when encryption is required. Regarding these issues, task-driven mechanisms that select the most relevant information (e.g. through visual attention) while smartly compressing it are required. These mechanisms are part of the active vision [CHI17] concept, which covers (among others) adaptation or smart compression.

Visual attention efficiently selects relevant features, and allows for data bandwidth reduction [BAR14]. The concept is taken from Biology, where perception is an active selection mechanism, where adaptation and compression plays an important role. Many visual attention models have been presented in the literature emulating the biological process, conjugating a bottom-up saliency and a top-down modulation pathways. The saliency mechanism selects areas based on how discriminative they are with respect to their environments. The top-down modulation biases the selection according to the performed task. This efficient mechanism has been applied in the past to many different fields such as robotics [FUJ10], autonomous navigation [LIU12], or military [CHE11].

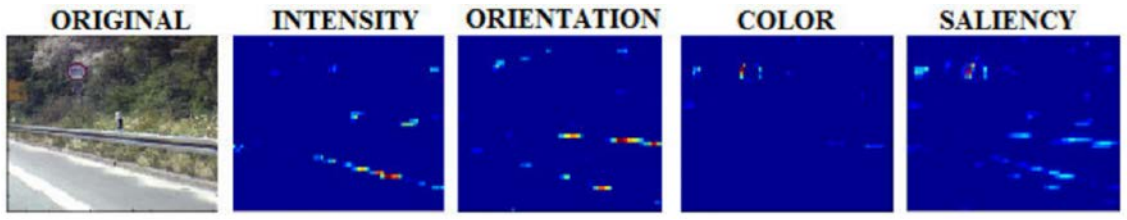


Figure 18. Example of saliency estimation from a driving scenario [BAR14]. Road mark, and the traffic sign are highlighted in the final saliency image (estimated with intensity, orientation, and color discriminative features).

The mechanisms will be applied to the Smart Grid and the Habit Tracking use cases. In the Smart Grid use case, the vision subsystem is focused on the video-surveillance of the perimeter of an electrical substation and the main functionality is the detection of suspicious behaviour and robust tracking of suspicious targets. In the Habit Tracking use case, the vision component will be focused on the classification of person behaviour indoors. Briefly, it consists on the understanding of human actions, determining its purpose and usually entails: a) feature extraction from video sequences and b) action classification fed with the features extracted in the first step to assign the correct label.

Two different strategies will be used: adaptation and selection of task-driven relevant features in order to reduce the data bandwidth and achieve real-time performance at the video processing components. The metrics upon which adaptation will be based depend on the application.

For the tracking application, accuracy and performance are the two qualities that will be taken into account. Accuracy can be defined as the deviation in pixels from the ground truth (real or labelled) location and the estimated location of the target. Performance is measured in number of processed frames per second. The adaptation will be based on the number of targets to track versus the expected performance in frames per second, ensuring real-time processing for up to 4-5 targets (we are not expecting many more targets for the considered use case application). However, we also expect the reduction of performance along with the increase of number of tracked targets.

To illustrate this with a real example, consider transmitting a video stream of 1024p images at 50 frames per second sums up to 150 MB/s. The application of an active scheme could bring it down to 3.7 MB/s by selecting a box of 64x64 (full resolution) for the target, and transmitting the original image using only 1 channel and scaled to 1/4 (256p), keeping the original frame rate.

Secondly, we will also implement a mechanism to adapt the resolution of the video feed to the saliency of the image data. Given a specific video source, more resolution and priority will be assign to that video source when a possible interesting target is detected in its field of view. Otherwise, low-resolution image transmission will ensure the sensible use of the data bandwidth and real-time processing.

For the behaviour classification, the actions to be studied are determined by the task. Potential actions identified at this point could be cooking, preparing coffee/tea, and actions that will trigger alarms such as accidental fall, fainting, or leaving the stove on (for the indoor scenario). The action classifier does not need to be run constantly, it requires a video sequence to do the action inference, which is just a label. With respect to the qualities to be used, Precision (fraction of positive labels that are correctly classified) and Recall (fraction of real positives that were correctly labelled) values are the most common ones in classification tasks. The adaption is this case correlates the number of actions to be classified (number of different labels, that determine the complexity of the classification method) and the performance (or actions labelled per second, or fixed number of frames).

Moreover, the number of visual features for the classification will also be adapted according to the number of classes. This will also determine the performance of the final network.

Although it is very seminal at this point, we are also considering the features that can be estimated at the node, and how to send only the relevant ones to the cloud to do the final processing. In this case, not only performance is considered; also the privacy, avoiding the transmission of full images to the cloud.

Our contributions will be packed in software libraries. Regarding tools, we are currently using C++ and OpenCV libraries, and GPU-based CUDA implementations to achieve real-time performance for our processing, running on Ubuntu systems.

4.4.3 Distributed Image Pre-Processing and Optimized Image Segmentation

Multiple view geometry is a complex and resource-demanding task. Thus, image pre-processing, such as undistorting and segmenting, has to be carried out in the most efficient way. Nonetheless, precision in the segmentation process is key to offer accurate results that truthfully represent the reality. Specially, when the application is focused on industrial inspection.

As pointed out by Shi et al. [SHI16], in a system where several images taken by a number of devices have to travel to a single node, an edge-computing approach can reduce latency and bandwidth usage, while increasing throughput. This approach is based on the principle that the workload should be finished in the nearest layer with enough computation capability to the things at the edge of the network. This translates into providing the cameras with computation capability in our envisioned application. We propose a distributed image processing pipeline where low-power execution boards are in charge of performing an initial image processing and a fast segmentation in order to increase throughput and reduce energy consumption.

Another approach for image processing distribution is based on the remote OpenCL-based software stack described in Section 3.1. A modification of this framework enables image processing pipelines to be distributed among several computational nodes located anywhere in the network. We were investigating the suitability of this approach for our foreseen final application (3D industrial inspection system), but this approach requires sending the images through the network, which increases bandwidth usage, rendering it a non-viable solution.

Therefore, we proposed a distributed image processing pipeline where low-power execution boards are in charge of performing an initial image processing and a fast segmentation in order to increase throughput and reduce energy consumption. These low-power boards are installed close to the cameras, thus, the first layer with computation capabilities is located immediately after images are captured.

The diagram in Figure 19 shows a typical configuration of this kind of system, where the number of low power execution boards and cameras can be decided at design time, while we include a new element, a 'dispatcher' to perform workload decisions at runtime.

The innovation proposed with our approach is twofold; first, until now, the 3D industrial

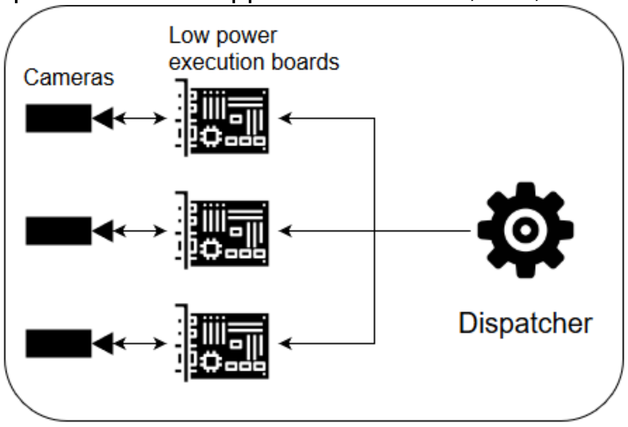


Figure 19. Typical configuration of an industrial inspection system.

inspection system we are enhancing in FitOptiVis relies on a single computing node where all the image processing takes place. Throughput can be increased with the distributed segmentation, as the low-power execution boards can segment new captures while the main computing node is working on the previous capture.

As a first step in our work, we evaluated several low-power execution boards to identify those most suitable in terms of computation power and cost trade-off. The table below shows the evaluated boards and their average segmentation time in milliseconds.

Table 1. Segmentation performance and cost of low-power execution boards

Board	**Segmentation time [milliseconds]**	**Cost (approx.) [euro]**
Espresso bin	μ = 1045.37; σ = 2.89	44€
Grapeboard	μ = 1839.74, σ = 23.88	190€
Raspberry Pi	μ = 3450.43; σ = 55.86	Discontinued
Raspberry Pi (optimised, neon flag)	μ = 2603.37; σ = 19.47	Discontinued
Raspberry Pi 3	μ = 1554.24; σ = 87.11	30€

Raspberry Pi 3 (optimised, neon and tune flags)	μ = 758.04; σ = 21.09	30€
Nvidia Jet-son TX2	μ = 130.78; σ = 5.26	500€

The times shown in Table 1 above were obtained using the same segmentation algorithm that is currently deployed on the main computing platform. No platform-specific optimizations were applied to the algorithm.

An example of one of the hardware configurations used is shown in Figure 20 below.

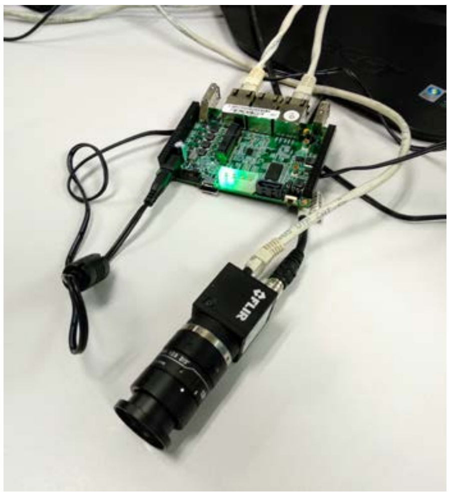


Figure 20. Marvell ESPRESSObin board connected to a HD camera.

To plan which phases of the segmentation process we should prioritize during the optimization of the algorithm on the most promising boards, we measured the average execution time of each phase of the algorithm on the low power execution boards. Table 2 below shows the fraction of time spent on each of the sub-tasks (only OpenCV related) in the segmentation process for the Espresso Bin board. The results are similar for the other boards tested.

Table 2. Fraction of execution time spent in different phases of the segmentation algorithm running on the Marvell EspressoBin board.

Background diff.	Blur	Erosion & Dilation	Finding contours	Gaussian Filter	Thresholding
3.63%	7.55%	29.04%	8.79%	47.37%	3.62%

Based on these results, the algorithm could benefit most from (platform-specific) optimizations in the 'Erosion & Dilation' and the 'Gaussian Filter' phases. To optimize these two phases, we are working on specific modifications of the OpenCV API to adapt to the particularities of the most promising boards. Specifically, we are working on removing floating-point computations and on a two-step segmentation process in which an initial (approximated) region of interest is found on a low-resolution image.

The second innovation that we achieve by employing low power execution boards installed close to the cameras is the reduction in bandwidth usage. Because images travel from the low-power boards to the main computing node already segmented, less bandwidth is used. As an example, the two images in Figure 21 below show a part processed by the system. The image on the left shows the raw data captured, while the image on the right shows the segmented image, reducing the total size of the image by about 30%. This kind of bandwidth reduction is extremely important, especially when using many cameras.

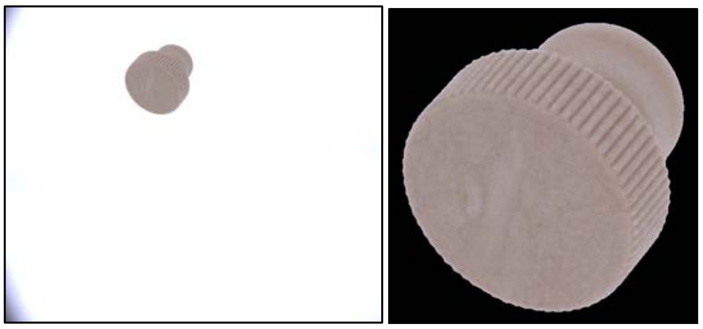


Figure 21. Example of image before (left) and after (right) segmentation

Moreover, low-power execution boards are capable of detecting incorrect captures and asking the capture system to retry a new capture of the same part. This decision is taken without the information travelling from the cameras to the computation cluster, which reduces bandwidth consumption even further.

All optimizations will be implemented in the segmentation algorithm in subsequent work, which will allow us to evaluate these innovations in comparison with the currently used approach. We are plan to focus on the optimization of the following metrics:

- **Average throughput**: this metric measures the number of parts processed per time unit. To obtain a relevant evaluation, a variety of tasks with different types of parts has to be employed. This variety should include parts that due to their shape are prone to produce incorrect captures.
- **Bandwidth usage**: this measures the number of bits per time unit that are transferred on average from the capturing devices to the main computation node.

The diagram in Figure 22 below shows the architecture of the system in which we plan to test and evaluate our optimizations.

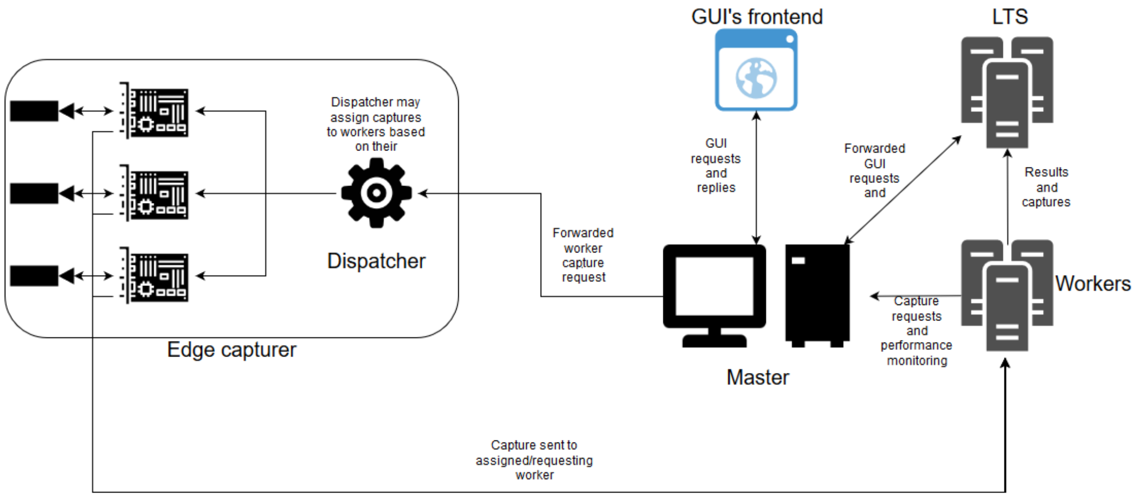

GUI's frontend
LTS
Dispatcher may assign captures to workers based on their
Dispatcher
Edge capturer
GUI requests and replies
Forwarded GUI requests and
Results and captures
Forwarded worker capture request
Workers
Capture requests and performance monitoring
Master
Capture sent to assigned/requesting worker

Figure 22. Architecture of the optimized capture system.

The workload dispatcher component (‘Dispatcher’) is a critical element for the runtime support. It will distribute the captured images among worker agents (‘Workers’). To this end, the dispatcher learns (at runtime) the average computation time that each worker requires to process a capture. Moreover, the dispatcher is able to recognize the type of part contained in the capture, because different part types require different checks and analyses and therefore the required processing time varies greatly. Using this information, the dispatcher delivers each capture to the worker that will contribute the most to the overall throughput of the system. The dispatcher component is currently under development. To evaluate the performance of the workload dispatcher we consider using average throughput.

Finally, we will integrate a performance monitor in the system so as to allow the operator to check and monitor the overall performance of the system. With the monitor, the operator will be able to detect malfunctioning hardware or incorrect configurations that are affecting the system, and take appropriate corrective measures.

4.4.4 Selective On-Demand Resource Loading

To achieve near real-time (soft real-time) performance on low-power mobile platforms, such as the HURJA’s Salmi Augmented Reality (AR) system, we plan to utilize smart feature extraction, segmentation, and classification algorithms to reduce bandwidth usage by only sending the necessary parts of images/videos.

Specifically in the context of the Salmi AR system, a mobile application called Extent can (upon request) download a JSON packet which consists of a list (descriptions) of wakeup images, objects, entities, and actions. Either the request can come from the Salmi MAPS website, from the Salmi AR mobile application, or directly from the Extent mobile application if the “free roam” state has been switched on (requires GPS). End-users have the option to switch the “free roam” state off at any time and when this happens, the Extent mobile application downloads new content only upon request from an external source (currently only the Salmi system related sources are available). The Extent mobile application downloads all required wakeup images, 3D-models, textures, audio files, videos, etc. based on the instructions received via JSON packet.

To optimize the run-time performance of the Salmi AR system, all of these packets can be downloaded in advance. All files will be saved locally into end-users' mobile device (smart phone or tablet) and those will be shown to end-users based on instructions received via JSON packets as soon as matching wakeup image, object, entity, or action has been found, or when an end-user is within a certain pre-defined distance from the target. Free roam data will be removed on-the-fly from end-users' devices when each session ends. The Extent mobile application is currently being developed using C# programming language on top of the Unity 3D engine and the server back-end side is currently being developed using PHP. During our early testing phase, all description packets are in JSON format.

The runtime state of the system includes measured performance and energy usage, which can be handled by a generic data model. Relevant metrics to be monitored/evaluated are the following:

- Near real-time (soft real-time) performance: System performance can be monitored/evaluated in terms of frames-per-second or kilobits-per-second, but AR-feature robustness/performance depends highly on the selected AR-glass model. We plan to start development with state-of-the-art Magic Leap and/or HoloLens 2 glasses to ensure that all possible use cases can be implemented easily. Later on we plan to investigate the use of other (cheaper and less powerful) AR-glass options that may require more optimization of the system code to achieve the level of performance comparable with the high-end, state-of-the-art AR-glasses.
- Optimal energy usage: It is not an easy task to calculate the initial energy usage for the whole Salmi AR system before the first MVP version is fully implemented, but continuous camera feed and required advanced algorithms will present a challenge in terms of optimizing the energy usage of the system as a whole. As soon as the first MVP version is ready, we will perform extensive measurements on power usage and based on the achieved results, we will make adjustments to the implemented algorithms to enable optimal energy usage of Salmi AR system.

In addition, the system monitors the achieved level of satisfaction of all end-user groups that can be handled by a generic data model:

- The intended users of the Salmi AR system will be brain damage patients (assisted living), elderly people (assisted living), relatives (monitoring and situational awareness), nurses (home visits), and doctors (emergency cases). We have made careful plans to achieve the required level of satisfaction for all of these end-users of our Salmi AR system. However, when our first MVP version will be ready by June 2019, we cannot yet completely fulfill all of the below-mentioned end-users requirements or all the needed features, but by the end of the project, we will have fully functional version of Salmi AR system that fulfils the level of satisfaction for all of these end-user groups.

4.4.5 Deterministic Networking for Time-Sensitive Data

Time Sensitive Networking (TSN) is an update of the IEEE Ethernet standard aimed to achieve synchronized and distributed control. TSN transports multi-purpose traffic providing differentiated Real-Time Quality of Service (RT-QoS). Protected traffic has guaranteed packet transport, in terms of bounded low latency, low packet delay variation and low packet loss.

The implementation of TSN for FitOptiVis is currently at a proof-of-concept stage. After a review of the latest revisions of the IEEE 802.3 and IEEE 802.1Q standards, a functional architecture and user APIs have been described. This solution will be applied to the Surveillance of smart-grid critical infrastructure and the Habit Tracking use cases.

The Surveillance of smart-grid critical infrastructure use case represents a distributed real-time control system. In this context, TSN provides the network backbone, transporting time-sensitive traffic between smart-grid, surveillance sub-networks in electrical substations, and remote central stations. Furthermore, accurate time synchronization is required to enable coherent processing and control monitoring.

The Habit Tracking use case requires hybrid communication between edge and cloud processing nodes. Each application will require traffic differentiation and RT-QoS for side-band communication between edge and cloud processing, as well as best effort bandwidth for processed data. In this context time synchronization will be sourced to support coherent processing of time-dependent signals and images. Furthermore, time synchronization between edge and cloud will enable performance monitoring.

The proposed TSN solution is devised for Xilinx Zynq-7000 platforms, which consist of a processing system (ARM-v9) and programmable logic. Thus, TSN functional modules are either software programs executing on the processor or IP-cores implemented in *gateware*. Configuration and monitoring can be achieved through corresponding APIs on each TSN module.

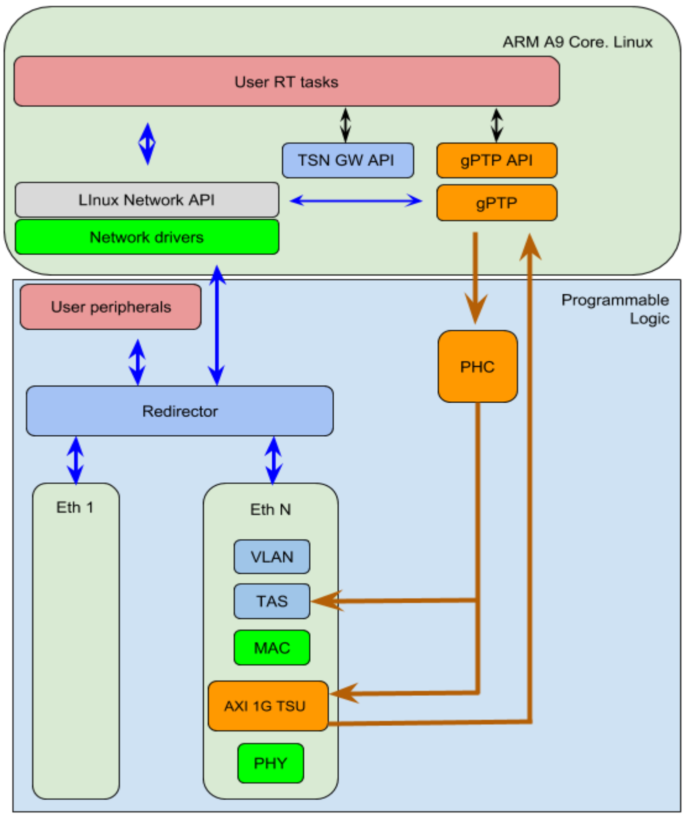


Figure 23. Architecture of a system implementing TSN.

The TSN system architecture, shown in Figure 23 above, can be split into two functional components:

- **The networking component**, which provides 1000 Base-T Ethernet connection, traffic differentiation and prioritization, in addition to priority-based, time-driven, strict arbitration of the output bandwidth. The blue modules, i.e., the redirector, the VLAN tagger (and untagger), and the Time-Aware traffic Shaper (TAS) implement the IEEE 802.1Q functionality, while the green modules, i.e., MAC, PHY, and DMA are off-the-shelf Xilinx IP-cores implementing standard IEEE 802.3 functionality. A Linux network driver is provided for this particular gateware.
- **The timing component**, which provides IEEE 802.1AS functionality, i.e. highly-accurate time synchronization among all TSN stations of the network. This component (orange modules) consists of a gPTP cyclic executive running on the PS and a PTP hardware clock, supported by TimeStamp Units (1G TSU's), present on each gPTP-capable interface.

Note that the TAS mechanism alone does not prevent interference between time-critical messages and lower-priority jumbo frames found in video streaming applications. This issue can be addressed considering guard bands on the output bandwidth cyclic schedule, at the cost of available bandwidth. To avoid potential RT-QoS violations and to optimise bandwidth usage, a frame pre-emption mechanism is being considered. Frame pre-emption is a MAC sublayer enhancement described on IEEE 802.3BR that stops lower priority frame transmissions whenever TAS notices a time-critical message should be transferred.

4.4.6 Algorithms and Techniques Envisioned to Achieve Real-Time Performance for PCC Demo System

With the selection of MPEG V-PCC as PCC encoding and distribution scheme, Nokia can utilise available video hardware decoders to carry the major load in the decoding process. Figure 24 illustrates the current V-PCC decoding scheme block diagram, where available hardware decoding support is marked in teal.

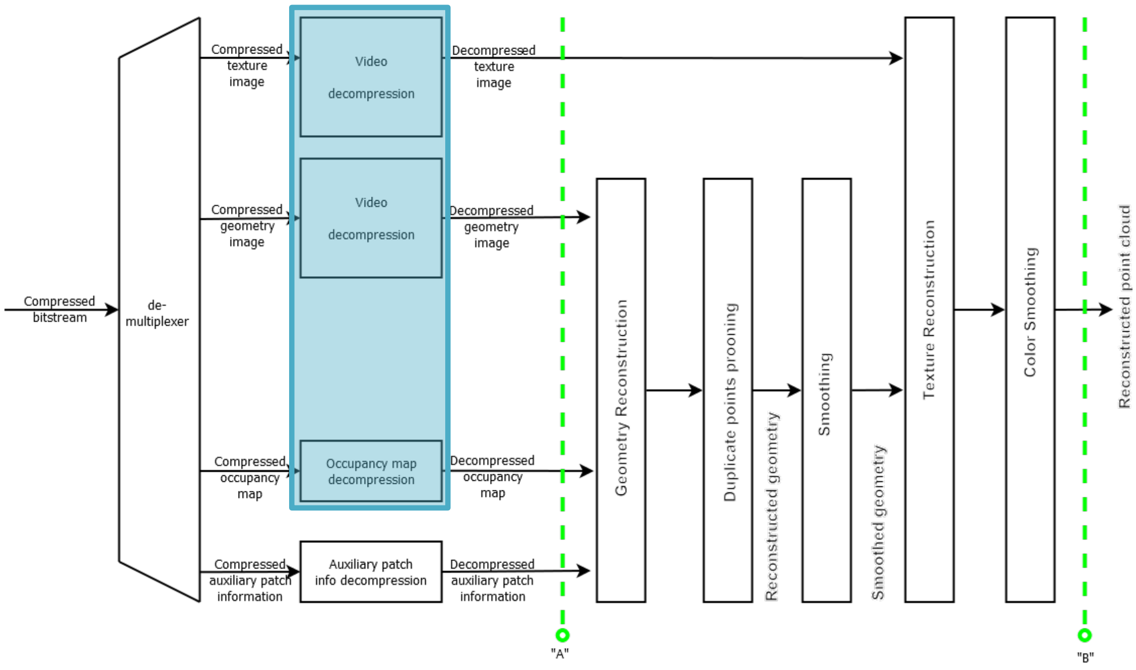

Compressed bitstream
de-multiplexer
Compressed texture image
Compressed geometry image
Compressed occupancy map
Compressed auxiliary patch information
Video decompression
Video decompression
Occupancy map decompression
Auxiliary patch info decompression
Decompressed texture image
Decompressed geometry image
Decompressed occupancy map
Decompressed auxiliary patch information
Geometry Reconstruction
Duplicate points prooning
Reconstructed geometry
Smoothing
Smoothed geometry
Texture Reconstruction
Color Smoothing
Reconstructed point cloud
"A"
"B"

Figure 24. V-PCC TMC2 decoding structure.

For the three decoding instances, texture, geometry and occupancy video decompression, not much special attention on real-time capability is required, as the current video decoding hardware can achieve much higher levels of decoding performance than demanded by the use case. However, attention is required due to three simultaneously running video decoder instances, which must be synchronised. This aspect and any possible implications must be further investigated within FitOptiVis.

As for the real-time decoding of auxiliary patch information, little is known so far, and detailed experiments have to be carried out to assess any implications on real-time performance, e.g. maximum number of patches per frame, inter-prediction between patch auxiliary information, random access structures, etc. This investigation will also be part of the planned FitOptiVis research topics.

Finally, decoding and rendering altogether has to happen in real-time. Thus, any unnecessary data transfers, e.g. copying 3D point cloud data from the CPU to the GPU for rendering, should be avoided. Therefore, we envision V-PCC decoding straight into the GPU memory, as well as tools for partial and simultaneous decoding and playback. Such tools, together with support of the hardware video decoders, should ensure real-time capability of our PCC demo system.

5. Conclusion

In our summary of the outcomes of Task 4.1 from the first year of the project, we deal primarily with two aspects of runtime support for adaptive applications developed using the FitOptiVis approach.

The first aspect concerns runtime platforms. To establish a consortium-wide awareness and agreement of platform components (as defined in deliverable D2.1) developed within the project, we presented an overview of the available runtime platforms. Each platform is suitable for different kind of applications, with requirements at different levels of abstractions, and operating at different time scales.

The second aspect concerns runtime adaptation and comprises two parts. The first part presented an overview of adaptation mechanisms and concepts supported by the runtime platforms. The second part presented an initial collection of adaptation scenarios from use-case owners participating in WP4. These will be used in the following year to steer development of the runtime platforms and interfaces for use by adaptive resource managers developed in the context of other WP4 tasks.

While a development version of some platforms (pocl-remote) had been already made available to project partners, other platforms are still not mature enough for consortium-wide release. In the following year, we will focus on making the runtime platforms available to partners in the project and provide assistance to partners targeting specific runtime platform components, and optimizing the scalability of the pocl-remote especially to reduce synchronization overheads in latency critical interactive applications. Where applicable, contributions to relevant open-source code bases will be made.

Also, with the reference architecture concepts from WP2 firmly in place, we will finalize the instantiations of the runtime platforms within the framework of the reference architecture to make the platforms amenable to tool support.

These activities will result in second iteration of this deliverable, which will incorporate outcomes from the second year of the project in deliverable D4.2.

References

[ADA17] O. Adam, Y. C. Lee, A. Y. Zomaya. CtrlCloud: Performance-Aware Adaptive Control for Shared Resources in Clouds. In Proc. CCGrid, IEEE, 2017, pp. 110–119.

[AMI17] M. Amiri, L. Mohammad-Khanli. Survey on prediction models of applications for resources provisioning in cloud. Journal of Network and Computer Applications 82 (2017)93–113.

[ARROW] Documents for Arrowhead Framework [Online]. https://forge.soa4d.org/docman/?group_id=58

[BAR14] Barranco, Francisco, Javier Diaz, Begoña Pino, and Eduardo Ros. "Real-time visual saliency architecture for FPGA with top-down attention modulation." IEEE Transactions on Industrial Informatics 10, no. 3 (2014): 1726-1735.

[BYS10] M. Bystrom, I. Richardson, S. Kannangara, and M. de-Frutos-Lopez. 2010. Dynamic replacement of video coding elements. Image Commun. 25, 4 (April 2010), 303-313.

[BUR01] W. Burleson et al., "Dynamically parameterized algorithms and architectures to exploit signal variations for improved performance and reduced power," in Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing. IEEE, 2001, pp. 901-904 vol.2.

[CHE11] Chen, Yen-Lin, Bing-Fei Wu, Hao-Yu Huang, and Chung-Jui Fan. "A real-time vision system for nighttime vehicle detection and traffic surveillance." IEEE Transactions on Industrial Electronics 58, no. 5 (2011): 2030-2044.

[CHE15] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, W. Knottenbelt. CloudScope: Diagnosing and Managing Performance Interference in Multitenant Clouds. In Proc. MASCOTS, 2015, pp. 164–173.

[CHE18] T. Chen, R. Bahsoon, X. Yao. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. ACM Comput. Surv. 51 (2018) 61:1–61:40.

[CHI17] Chittka, Lars, and Peter Skorupski. "Active vision: a broader comparative perspective is needed." (2017).

[DEL13] C. Delimitrou, C. Kozyrakis, Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In Proc. ASPLOS, ACM, 2013, pp. 77–88.

[DEL14] C. Delimitrou, C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In Proc. ASPLOS, ACM, 2014, pp.127–144.

[FAN15] F. Faniyi, R. Bahsoon. A Systematic Review of Service Level Management in the Cloud, ACM Comput. Surv. 48 (2015) 43:1–43:27.

[FUJ10] Fujita, Toyomi, Kazuya Chiba, and Claudio Privitera. "Bottom-up regions-of-interest in observation of robot hand movement: Comparisons with humans." IEEE Int. Conf. on Systems, Man and Cybernetics, pp. 3768-3773, 2010.

[GAR14] M. García-Valls, T. Cucinotta. C. Lu. Challenges in real-time virtualization and predictable cloud computing. Journal of Systems Architecture 60 (2014) 726–740

[GOV11] S. Govindan, J. Liu, A. Kansal, A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In Proc. SOCC, ACM, 2011, p. 22.

[MAR11] J. Mars, L. Tang, R. Hundt, K. Skadron, M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In Proc. MICRO, ACM, 2011, pp.248–259.

[GOO17] Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, van Kampenhout, Reinier, Rasool Tavakoli, Juan Valencia, Ahmadi Balef, Hadi, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi:. "NOC-Based Multi-Processor Architecture for Mixed Time-Criticality Applications", In Handbook of Hardware/Software Codesign, Springer, 2017.

[HAM16] A. Hameed, A. Khoshkbarforoushha, R. Ranjan, P. P.Jayaraman, J. Kolodziej, P. Balaji, S. Zeadally, Q. M.Malluhi, N. Tziritas, A. Vishnu, S. U. Khan, A. Zomaya. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. Computing 98 (2016) 751–774.

[KAD18a] Jiří Kadlec, Zdeněk Pohl, Lukáš Kohout: “Design Time and Run Time Resources for the ZynqBerry Board TE0726-03M with SDSoC 2018.2 Support”, [Online]. http://sp.utia.cz/index.php?ids=projects/fitoptivis

[KAD18b] Jiří Kadlec, Zdeněk Pohl, Lukáš Kohout: “Design Time and Run Time Resources for Zynq Ultrascale+ TE0820-03-4EV-1E with SDSoC 2018.2 Support”, [Online]. http://sp.utia.cz/index.php?ids=projects/fitoptivis

[KAD18c] Jiří Kadlec, Zdeněk Pohl, Lukáš Kohout: “Design Time and Run Time Resources for Zynq Ultrascale+ TE0808-04-15EG-1EE with SDSoC 2018.2 Support”, [Online]. http://sp.utia.cz/index.php?ids=projects/fitoptivis

[KAD18c] Jiří Kadlec, Zdeněk Pohl, Lukáš Kohout: “Design Time and Run Time Resources for Zynq Ultrascale+ TE0808-04-15EG-1EE with SDSoC 2018.2 Support”, [Online]. http://sp.utia.cz/index.php?ids=projects/fitoptivis

[KEP03] Kephart, J., Chess, D.: The Vision of Autonomic Computing. Computer. 36, 1, 41–50 (2003).

[KOH18] Lukáš Kohout, Jiří Kadlec, Zdeněk Pohl: “Video Input/Output IP Cores for TE0820 SoM with TE0701 Carrier and and Avnet HDMI Input/Output FMC Module”, [Online]. http://sp.utia.cz/index.php?ids=projects/fitoptivis

[LIU12] Liu, Zhong, Weihai Chen, Yuhua Zou, and Xingming Wu. “Salient region detection based on binocular vision.” IEEE Conference on Industrial Electronics and Applications (ICIEA), pp. 1862-1866. IEEE, 2012.

[MAN15] Z. A. Mann. Allocation of Virtual Machines in Cloud Data Centers—A Survey of Problem Models and Optimization Algorithms. ACM Comput. Surv. 48 (2015)11:1–11:34.

[NAT10] R. Nathuji, A. Kansal, A. Ghaffarkhah. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. Proc. EuroSys 2010, ACM, 2010, pp. 237–250

[SAT17] M. Satyanarayanan. The Emergence of Edge Computing. Computer 50 (2017) 30–39.

[SIN15] S. Singh, I. Chana. QoS-Aware Autonomic Resource Management in Cloud Computing: A Systematic Review. ACM Comput. Surv. 48 (2015) 42:1–42:46.

[TE0726] Trenz Electronic: "TE0726 TRM", [Online]. https://shop.trenz-electronic.de/en/27229-Bundle-ZynqBerry-512-Mbyte-DDR3L-and-SDSoC-Voucher?c=350

[TE0701] Trenz Electronic: "TE0701-06-Carrier-Board" [Online]. https://shop.trenz-electronic.de/en/TE0701-06-Carrier-Board-for-Trenz-Electronic-7-Series?c=261

[TE0808] Trenz Electronic: "UltraSOM+ MPSoC Module with Zynq UltraScale+ XCZU15EG-1FFVC900E, 4 GB DDR4", [Online]. https://shop.trenz-electronic.de/en/TE0808-04-15EG-1EE-UltraSOM-MPSoC-Module-with-Zynq-UltraScale-XCZU15EG-1FFVC900E-4-GB-DDR4?c=450

[TE080X] Trenz Electronic: "UltraITX+ Baseboard for Trenz Electronic TE080X UltraSOM+" [Online]. https://shop.trenz-electronic.de/en/TEBF0808-04-UltraITX-Baseboard-for-Trenz-Electronic-TE080X-UltraSOM?c=261

[TE0820] Trenz Electronic: "MPSoC Module with Xilinx Zynq UltraScale+ ZU4EV-1E, 2 Gbyte DDR4 SDRAM, 4x5cm", [Online]. https://shop.trenz-electronic.de/en/TE0820-03-04EV-1EA-MPSoC-Module-with-Xilinx-Zynq-UltraScale-ZU4EV-1E-2-Gbyte-DDR4-SDRAM-4-x-5-cm

[XU18] R. Xu, S. Mitra, J. Rahman, P. Bai, B. Zhou, G. Bronevetsky, S. Bagchi. Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Colocated Workloads. In Proc. Middleware, ACM, 2018, pp. 146–160.

[YAN13] H. Yang, A. Breslow, J. Mars, L. Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In Proc. ICSA, ACM, 2013, pp.607–618.

A. Review of Virtualization and Resource Management Techniques

This appendix provides a review of the state-of-the art in the area of virtualization and resource management techniques.

A.1 State-of-the-art in Virtualization Techniques

Virtualization refers to the abstraction of a physical component into a virtual object whereby a greater measure of utility can be obtained from the resource component offers [1]. From the hardware perspective, virtualization refers to the abstraction of computer resources whereby applications are decoupled from the hardware they execute on. While the virtualization concept has been there since 1960s, when IBM developed virtualization to enable concurrency by partitioning a mainframe into logical machines [2], it has gained extra attention in the past decade possibly due to the proliferation of cloud services. The main advantages of virtualization are:

- Consolidation: Consolidation refers to bringing together separate parts into a single or unified whole. Virtualization enables consolidation by bringing together several under-utilized execution platforms (i.e., machines) into a single execution platform, thereby reducing operating costs. This has been commonly referred to as multi-tenancy in the literature.
- Isolation: Virtualization enhances security as well as reliability by providing isolated environments where applications running in one virtual execution platform cannot affect applications running in another one. Regarding the security, less-trusted applications can be executed in separate virtual execution platforms, thereby preventing them from accessing and affecting other applications. Virtualization improves the reliability by providing isolated environments where faults and bugs in one environment cannot interfere with other environments.
- Flexibility: Virtualization provides flexible environments for applications where their allocated resources can change dynamically in response to changes in their demands. This includes modifying both the amount of resources and the mapping of virtual resources to physical ones. They are commonly called elasticity and live migration in the literature [3].

Although there are several types of virtualization (such as application virtualization, network virtualization, storage virtualization, etc.), we focus on platform virtualization (also called hardware virtualization or system virtualization in general, and server virtualization in cloud-oriented papers). By platform virtualization, we mean adding a layer between applications and the underlying hardware (called virtualization layer) which creates virtualized environments for applications to be deployed on. Based on the type of this layer, we classify the existing techniques into two classes, namely hypervisor-based virtualization and container-based virtualization, which are elaborated upon in the following sections.

A.1.1 Hypervisor-based Virtualization

For a long time, the term virtualization was used only for hypervisor-based virtualization. The hypervisor, also called Virtual Machine Monitor (VMM), is a software that abstracts the underlying hardware into virtual components called Virtual Machines (VMs). Since the VMs need a complete execution platform (made up of various resources) to run, the hypervisor must virtualize all the underlying hardware resources (such as CPU, memory, storage, and I/O devices). The underlying hardware where the hypervisor runs is usually called the host, and the VMs that run on top of the hypervisor are called guests. Similarly, the operating system that runs on the host is called the host operating system, and the one running in a VM is called the guest operating system.

Based on the presence of the host OS, hypervisors are categorized into two classes, namely Type-1 (also called native or bare-metal) hypervisors and Type-2 (also called hosted) hypervisors. As their names imply, Type-1 hypervisors run directly on the hardware and have their own drivers, whereas Type-2 hypervisors run on top of a host OS and need its facilities to perform their tasks. The most well-known Type-1 hypervisors are:

- VMWare ESX Server [4]
- Microsoft Hyper-V [5]
- Xen [6]
- L4 microkernel family
- CoMik [7]
- XtratuM [8]
- PikeOS [9]

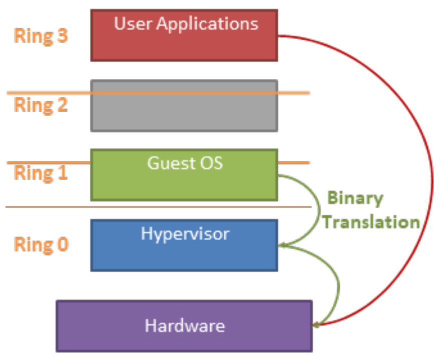
The examples of Type-2 hypervisors include but not limited to:

- Vmware Workstation and Vmware Player [10]
- VirtualBox [11]
- Parallels Desktop for Mac [12]
- QEMU [13]
- KVM [14]

Virtualization using Type-2 hypervisors is more suitable for enabling single users or small organizations to run VMs on a single machine. However, when high performance virtualization strategies are demanded, virtualization using bare-metal hypervisors, which impose less overhead due to direct interaction with the hardware, are more appropriate. Hypervisor-based virtualization approaches can be further classified into four categories which are explained next.

Full Virtualization

In full virtualization, the hypervisor emulates all hardware resources on the virtual system, allowing for running unmodified guest operating systems in VMs. One of the key components that must be emulated in this method is the processor's instruction set architecture. When operating systems run within VMs, they are not privileged enough to execute privileged instructions for interrupt handling, reading and writing to devices, and virtual memory.



Full virtualization

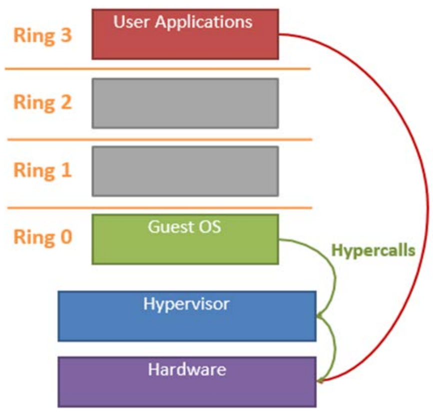
For instance, on the x86 architecture, there are four privilege levels (also known as rings) where the components running in level 0 are the most privileged, and the ones executing in level 3 are the least privileged. Usually, in non-virtualized systems, operating systems execute at level 0, and user applications execute at level 3. Unlike the normal instructions (e.g., ADD, SUB, etc.), the privileged instructions (e.g., HLT, invalidate a TLB entry, etc.) can only be executed by the components running in level 0. As shown in figure, in a virtualized environment, guest operating systems execute in level 1, which inhibits them from executing privileged instructions.

Since guest operating systems are unaware that they are running in a virtualized environment, they try to execute the privileged instructions similar to the case where they run in level 0. However, these attempts result in creating traps that go into the hypervisor which then emulates the expected functionality. Therefore, the guest OS never knows that it is running in a VM. Note that the non-privileged instructions execute directly on the hardware without the intermediation of the hypervisor. This technique is called trap and emulate.

However, there are some thorny issues with this technique. In some architectures, some privileged instructions may fail silently (which are sometimes called virtualization holes). For example, some instructions execute both in the privileged mode and non-privileged mode. However, they produce different results depending on the execution mode. To overcome this issue, a common approach called binary translation is used by the hypervisor. In this approach, the hypervisor scans the unmodified operating system binaries and modifies the offending instruction sequences, making sure that they are dealt with carefully. Since every privileged instruction results in a trap into the hypervisor, the full virtualization method can cause significant performance loss in some workloads. The most well-known products that perform full virtualization are Vmware Workstation, Microsoft Virtual Server, VirtualBox, Parallels Desktop for Mac, and QEMU.

Para-virtualization

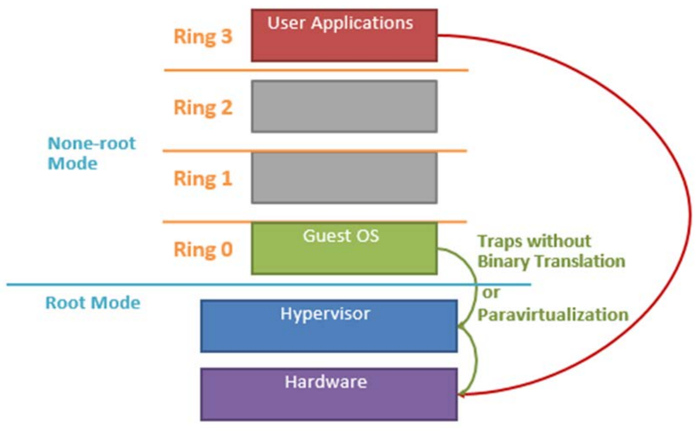
Para-virtualization (also known as OS-assisted virtualization) is an alternative approach to perform the virtualization. In this approach, the guest operating system is modified such that it is aware of being running within a VM. That is, as shown in figure, privileged instructions (i.e., non-virtualizable instructions) are replaced by calls to the hypervisor (also known as hypercalls). Therefore, compared to the full virtualization where the communication from the guest operating system to the hypervisor is always implicit via traps, in para-virtualization, the communication is explicit via hypercalls. This can offer performance improvements compared to the full virtualization for some workloads. However, since the guest operating system needs to be modified, it causes compatibility and portability issues. Note that the para-virtualization does not require any changes in Application Binary Interfaces (ABIs). Hence, the applications running on top of guest operating systems do not need any modifications. The most notable hypervisors performing para-virtualization are Vmware ESX, OKL4, XtratuM, and Xen.



Para-virtualization

Hardware-assisted Virtualization

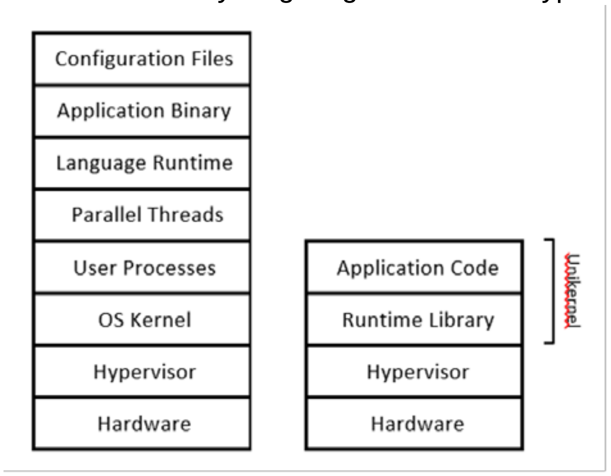
In hardware-assisted virtualization (also known as accelerated virtualization), the underlying hard-ware provides facilities to accelerate the execution of VMs. For instance, as shown in figure, a new CPU privilege mode (called root-mode) has been added to x86 processors since 2006 whereby privileged calls are automatically trapped to the hypervisor without needing to perform binary translation or para-virtualization. These virtualization extensions are introduced in Intel VT-x and AMD-V technologies for Intel and AMD processors respectively. Since the guest operating systems are not modified in hardware-assisted virtualization, it is similar to full virtualization to some extent. However, given the fact that binary translation is not required anymore, hardware-assisted virtualization is considered to be a faster approach. Note that hardware-assisted virtualization is not supported in older systems. The hypervisors that leverage hardware-assisted virtualization include, but are not limited to Vmware ESX, KVM, Hyper-V, and Xen.



Hardware assisted

Unikernels

Unikernel technology emerged in 2013 with the development of MirageOS [15]. The aim was to create specialized, single-purpose VMs whose unnecessary functionalities are removed at compile time, thereby reducing the footprint of an application running in the cloud. Unikernels are based on library operating systems proposed in the past (e.g., Exokernel [16] and Nemesis [17]). However, the hardware compatibility problems faced by these library Oss are solved in unikernels by targeting a standard hypervisor. As shown in figure, during the creation of unikernels, the required system libraries, language runtime, application binary, and configuration files are compiled into a single-address-space VM which runs directly on a standard hypervisor. Accordingly, the scheduling and resource management of unikernels are done by the hypervisor. Note that since there is only one address space, context switches between user and kernel space are not needed anymore, which results in a better performance compared to the traditional VMs. In other words, both the application and kernel components run at the privilege level 0, which is not optimal in terms of security isolation [18]. Although unikernels were first introduced for cloud applications, their lightweight nature has made them a promising solution for upcoming IoT edge applications [19].



Comparison of software layers in traditional VMs and unikernels

The most notable unikernel implementations include:

- MirageOS [15]
- HaLVM [20]
- Osv [21]
- IncludeOS [22]
- ClickOS [23]

A.1.2 Container-based Virtualization

Container-based virtualization (also known as operating system virtualization or containerization) aims at virtualizing the OS kernel rather than the hardware. It is usually considered as a lightweight alternative to hypervisor-based virtualization. The main difference between hypervisor-based virtualization and containerization is that in the former, each VM has its own OS kernel, while in the containerization, all the containers share a single kernel. Hence, containers are more lightweight than VMs. However, hypervisor-based solutions provide more flexibility by enabling the running of multiple operating systems on a single machine. A container image contains an application plus all its dependencies, libraries, and configuration files. A container is a runnable instance of a container image, which essentially is a group of processes that are isolated from other containers or processes in the system. The OS kernel (or container engine in particular) provides this isolation. Being light-weight in nature, containers are becoming the predominant technology in resource-constrained environments such as edge- and fog-based systems [24]. The examples of containerization solutions include, but are not limited to:

- Linux Containers (LXC) [25]
- Ubuntu LXD [26]
- Windows Containers [27]
- Docker [28]
- OpenVZ [29]
- BSD Jails [30]
- Solaris Zones [31]

Since the Linux-based solutions are more common in embedded/IoT architectures, Linux containers have been more focused on. Containers in Linux are realized by leveraging two kernel features, namely control groups and namespaces. Control groups (also called cgroups) is a kernel feature that limits, accounts for and isolates the CPU, memory, disk I/O and network's usage of one or more processes. On the other hand, a cgroup is a set of processes that are bound to a set of limits defined by the cgroup filesystem. Namespaces allow for isolation of global system resources between independent processes, and they provide processes with their own system view. Processes within a namespace only see processes in the same namespace. This type of isolation prevents groups of processes from manipulating other groups. Linux provides several namespaces to isolate system resources such as process identifiers (PIDs), filesystem mount points, and network devices, to name but a few.

A.1.3 Comparison

From the previous discussions on virtualization techniques we can conclude that each approach has its own advantages and disadvantages, which makes it impossible to designate a single approach the perfect solution for virtualization. Accordingly, in this section, we compare the aforementioned techniques from various aspects. Quite a few works exist in the literature that compare virtualization techniques. Hence, to begin with, we review a group of these publications, and subsequently, we summarize the outcomes of these works.

Literature Review

A detailed performance comparison of hypervisor-based virtualization and recently proposed lightweight solutions (including the containers and unikernels) is presented in [32]. Using a number of benchmarking applications, the authors compared four virtualization solutions, namely KVM (as a hypervisor-based approach), LXC and Docker (as containerization approaches), and Osv as a unikernel approach. The considered performance aspects include CPU, Memory, Disk I/O, and Network I/O performance. The measurements show that dominance of a virtualization solution is not necessarily consistent in all the applications. For instance, in two disk performance experiments, LXC performs better than Docker in one experiment, and in the other one, the results are the other way around. However, it can be generally stated that containers outperform VMs in roughly all the experiments. For instance, containers achieve near-native performance for disk intensive benchmarks, while KVM's throughputs for disk write and read are approximately a third and a fifth of the native run, respectively. Since the unikernel approach is not included in all the experiments, we cannot reach any conclusions about its performance compared to others. Nevertheless, they have shown that in memory performance experiments, unikernels perform worse than containers and VMs, and in the network performance experiments, they perform better than VMs but worse than containers.

The work presented in [33] compares four hypervisors (Hyper-V, KVM, vSphere, and Xen) under hardware-assisted virtualization settings in different use cases. The most important outcome of the work is that none of the hypervisors has been found superior to the others. Accordingly, effective management of hypervisor diversity with the goal of matching applications to the best platform is a significant challenge. The authors point out that a cloud environment should support different software and hardware platforms to meet various requirements. The authors have also performed experiments to measure interference caused by multiple tenants, showing that Hyper-V is sensitive to CPU, memory, and network interference. For KVM, although the response times are highly variable, none of the interfering benchmarks considerably degrade the performance. vSphere is highly sensitive to memory interference, while its sensitivity to CPU, disk, and network interference is very low. Finally, Xen’s interference sensitivity on memory and network is relatively high compared to the other hypervisors. These results also support the fact that there is no dominant hypervisor with superior performance in all circumstances.

The work presented in [34] evaluates the effects of multi-tenancy on the performance of different virtualization technologies (VMs and containers) in data center environments. The authors compare LXC containers and KVM virtualization and the results show that in general, the interference caused by co-located applications is more severe in the case of containers. In the case of single-tenant scenario, LXC performance is near the performance of bare-metal execution. On the other hand, KVM imposes high performance overhead in case of I/O intensive applications. In case of co-located applications (i.e., multi-tenancy), the results for CPU intensive workloads show that containers are more susceptible to interference. However, in memory-intensive workloads, containers offer acceptable isolation, whereas KVM performs better. In disk I/O isolation experiments, the latency increases by a factor of 8 for LXC, which implies the poor disk isolation in containers. Since the disk I/O performance is not high for VMs even in the isolated cases (and therefore enough bandwidth is available for other VMs), the latency increases only two times for KVM. These measurements demonstrate that isolation is stronger in VMs. Additionally, the authors have studied the impact of virtualization solutions’ capabilities on the management and development of applications. In particular, they show how the different characteristics of containers and VMs affect their management in a cluster. From the resource allocation perspective, since VMs somehow share the raw hardware, the resource allocation is also in that granularity (e.g., a fixed number of virtual CPUs). However, in the case of containers, resource control knobs offered by the OS (e.g., CPU scheduling) are more varied, which adds more dimensions to resource allocation. In other words, the resource allocation for containers involves allocation of both physical and OS resources. They also point out that dynamic resource allocation in VMs is fundamentally a hard problem, on the grounds that their virtual hardware is allocated before boot-up, and dynamically change their resource during execution requires “device hotplug” support by the guest OS. However, soft limits in containers provide a dynamic resource allocation mechanism, thereby achieving better performance on overcommitted hosts. Additionally, the authors compared VMs and containers from migration perspective. It is stated that unlike VM migration which is mature and widely used in data centers, container migration is more challenging and not mature yet. Another comparison between VMs and containers made in this work is comparison of their images. It is shown that for the same applications, container images are considerably smaller and faster to construct, which enables faster deployment and lower storage overhead.

Several other works exist in the literature which perform such experiments to compare the virtualization techniques and solutions; [35] compares Xen and KVM; [36] compares KVM and Docker; [37] compares Xen, OpenVZ, and XenServer; [38] performs a comparison between Xen, KVM, VirtualBox, and VMWare ESX; [39] compares software and hardware techniques for x86 virtualization; and [40] presents a comparison between VMs, containers, and unikernels; to name but a few. Additionally, a survey of container-based performance evaluation is conducted in [41]. However, the outcomes of these works are in line with what we discussed above and we therefore do not review them here.

Summary and Conclusions

Based on the technique used to perform virtualization, the virtualized environment is called VM, container, or unikernel. Figure 25 compares the structure and layers of these virtualized entities. Two key points can be inferred from this figure:

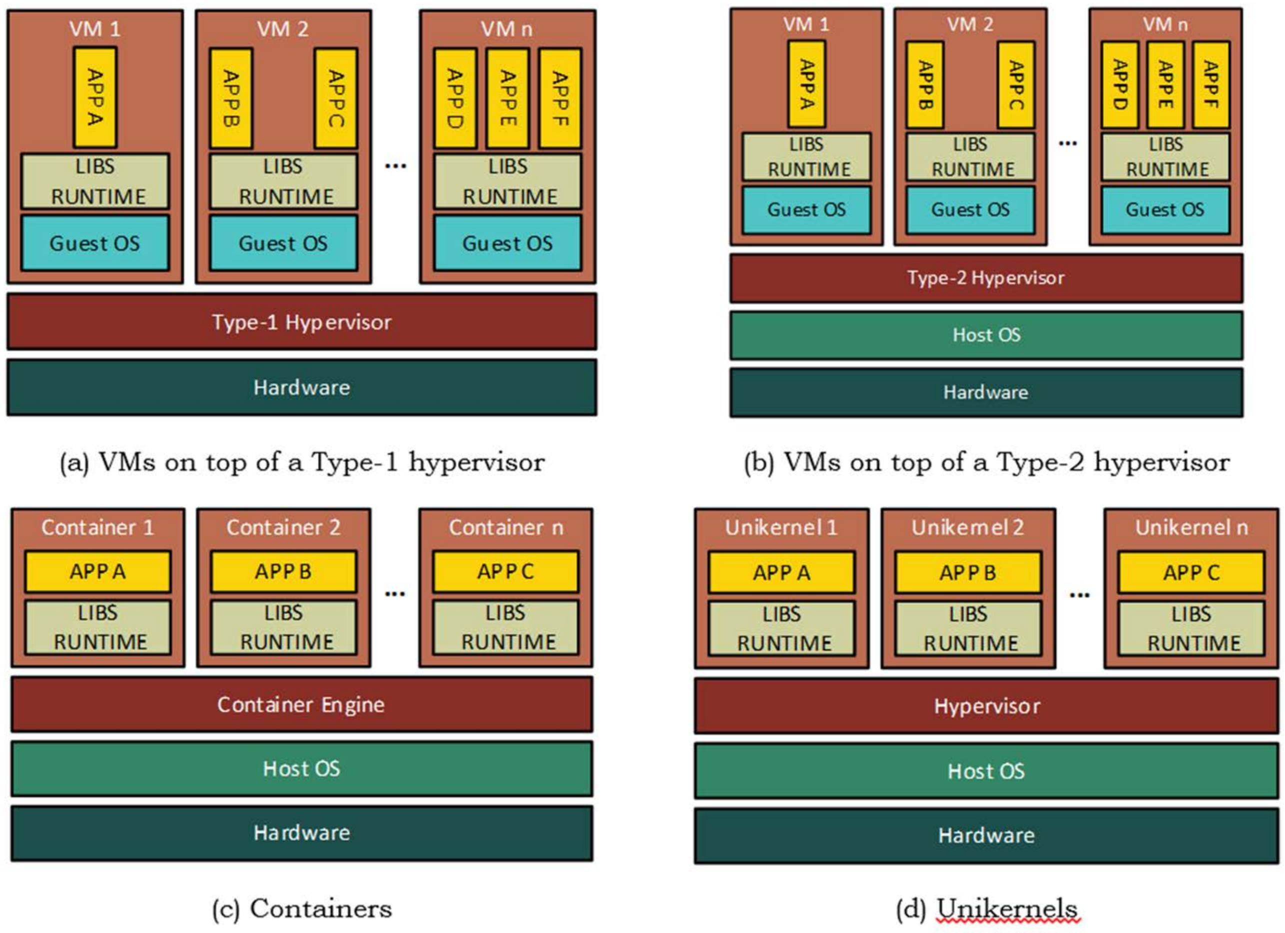


Figure 25. Structural comparison of virtualization solutions.

- Container and unikernels do not have a complete guest OS in their software stack, making them lighter than VMs.
- VMs are used to isolate complete systems – including an OS and a number of applications running on top of it – whereas containers and unikernels are employed to isolate applications.

Furthermore, we can draw an important conclusion from the results of prior works on the comparison of virtualization techniques which is the lack of a predominant virtualization solution performing better than other solutions in every circumstance. Even within a technique (such as hypervisor-based technique), each solution can only outperforms others in a few aspects, but never in all. Accordingly, to demonstrate the trade-offs between virtualization solutions, we summaries the outcomes of the prior works in Table 3.

Table 3. Comparison of virtualization solutions characteristics.

Virtualization Technology	Image Size	Boot Time	Memory Usage	Isolation	Flexibility in Resource Management	Performance	Programming Language Restrictions	Live Migration Support
Virtual Machine	~1000 MBs	~3-10 s	~100 MBs	High	Low	The worst	No	Yes
Container	~50 MBs	~<1 s	~5 MBs	Low	High	The best	No	Yes (not mature yet)
Unikernel	~<10 MBs	~<40 ms	~10 MBs	High	Low	Better than VMs, worse than containers	Yes	No

A.2 State-of-the-art in Resource Management

In a computing infrastructure, at any instance of time, resources must be effectively allocated to applications in such a way that their quality requirements are met. The dynamic nature of applications, which implies fluctuations in their resource demands, and the limited amount of available resources, which indicates that resources must be shared among applications, complicate the resource management process.

Although the infrastructure where resource management is performed span cloud infrastructures to stand-alone devices, in this work, we narrow our focus on resource management in fog/edge environments. Hong et al. [24] argue that resource management in fog/edge environments is challenging, since the applications compete for the resources which have limited capacity (e.g., limited power budget) and are heterogeneous (e.g., processors with different architectures), and their workloads change dynamically. Additionally, they argue that the cloud computing model is not practical for using in this paradigm, because it is likely to increase communication latencies when scores of devices are connected to the Internet. Consequently, applications will be adversely impacted because of the increase in communication latencies, and the overall Quality of Service (QoS) and Quality of Experience (QoE) will be degraded. Before getting into further discussions, it is worthwhile to make a distinction between the edge computing and the fog computing paradigms:

- A computing model that makes use of resources located at the edge of the network is referred to as "edge computing " [42]. Note that there is no single accepted definition of "edge" in the literature. There exists a broad definition "anything that's not a traditional data center could be the 'edge' to somebody" [43], which implies that edge of the network is somewhere nearer than data centers to the requestors.
- A model that makes use of both resources located at the edge of the network and the cloud is referred to as "fog computing" [44].

In order to study the literature, we review the following aspects of existing resource management frameworks.

A.2.1 Resource Types and Models

As discussed earlier, in the fog computing model, resources located both at the cloud and the edge of network are used to form a computing environment. These resources can be categorized under four resource types, namely compute resources, networking resources, storage resources, and power resources. In the cloud context, compute resources are a set of Physical Machines (PMs) that are usually partitioned into several virtual machines using techniques. Each physical machine has one or more CPUs, memories, network interfaces, and I/O devices. However, most of the works only consider processing and memory capacity in their compute resource models [3]. The PMs located at the cloud must be interconnected with a high-bandwidth network. It is shown that the overall performance of cloud services is governed by the communication overhead of PMs [3], which emphasizes the importance of managing the network resources within a cloud infrastructure. Storage services provided by public cloud providers (e.g., Amazon) include various types ranging from virtual disks and database services to object stores [3]. In the cloud infrastructures, the power consuming components are servers, networking equipment, power distribution instruments, cooling appliances, and supporting infrastructure. It is estimated that energy costs account for 42% of the overall operational costs in data centers [47]. Although devising low-power hardware components and efficient application implementations can reduce these costs, power-aware resource management can substantially contribute to total cost reduction as well. A survey of such power-aware resource management techniques for cloud computing systems is presented by Hameed et al. [48].

On the other hand, in the edge computing context, Single Board Computers (SBC) and commodity products comprise the compute resources [24]. SBCs (e.g., Raspberry Pi) are small computers containing processors, memory, network, and storage devices. They have been used in some works as fog/edge nodes [49, 50]. Besides the SBCs, commodity products (e.g., laptops and smartphones) are also employed as fog/edge nodes. Networking resources (i.e., network devices) for fog/edge computing are comprised of gateways and routers, WiFi Access Points (APs), and edge racks [24]. Hong et al. [51] have proposed an approach where under-utilized laptops (resources from public crowds), desktops at the edge of the network, and servers in the cloud are utilized to execute an animation rendering service. They have proposed a prediction method based on machine-learning techniques to predict the completion time of rendering jobs according to available resources. Using a motivational example, they have demonstrated that GFLOPS (Giga Floating Point Operations per Second) is not enough to abstract computation power. Other factors such as number of cores and clock frequency must be included in the model as well. To train their prediction models, they have used datasets where CPU, RAM, disk, and network resources have been considered [52]. The budgets are described in GHz and number of cores for CPUs, GB for memories, read/write throughputs in MB/s for disks, and receive/transmit rates in MB/s for network resources.

Noreikis et al. [53] have proposed a capacity planning solution for hierarchical edge cloud consisting of edge nodes and public clouds that considers QoS requirements in terms of response delay, and diverse demands for CPU, GPU, and network resources. CPU and GPU budgets are described in utilization percentage, and network budgets for transmission and receiving are expressed in KBps, indicating the network speed. Chen et al. [54] have proposed an offloading framework—called HyFog—that accounts for device-to-device and cloud offloading techniques. They have used CPU cycles per unit time to describe compute capacity, and the network links (including cellular links and device-to-device links) are abstracted using download/upload data rates and transmission/receiving power. Wang et al. [55] have proposed the ENORM (Edge NOde Resource Management) framework that realizes fog computing by integrating the edge of the network in the computing environment. They propose a provisioning and deployment mechanism to integrate an edge node with a cloud server. The proposed framework provisions CPU and memory to users. They are described in terms of the number of resource units each of which is one core of CPU and 200MB of RAM.

Liu et al. [56] have proposed an edge computing framework—called ParaDrop—which is implemented on WiFi Access Points or other wireless gateways. In the resource management part of ParaDrop, the controlled resources include CPU (expressed in CPU shares), memory (expressed in maximum allowed memory), and networking (traffic shaping is used to restrict the bandwidth). A fog computing architecture has been proposed by Gu et al. [57] which uses VMs for a medical cyber-physical system (MCPS). The proposed architecture utilizes computational resources in the network edge (e.g., base stations) to store and analyze the health information collected from low power sensors and actuators. Their research investigates the QoS guaranteed minimum cost resource management in fog computing supported MCPS. It is stated that the framework manages the computation capacity of base station resources; however, the resource types and models are not reported. An elastic real-time surveillance system architecture is proposed by Wang et al. [58] where surveillance cameras send images to a distributed edge cloud platform. The proposed system launches Virtualized Network Functions (VNFs) on the edge servers to execute data processing tasks. Resources are provisioned using VMs where resources are described by the number of vCPUs, size of RAM, and size of storage. Morabito et al. [59] have proposed the design of an Edge Computation Platform which leverages container-based virtualization technologies to build an environment for IoT applications. They use single board computers to create smart gateways whose CPUs, GPUs, and storage resources are being managed. There are no discussions on resource models.

An architectural framework—called Foggy—is proposed by Santoro et al. [60] which offers the functionality of negotiation, scheduling, and workload placement considering resource requirements (e.g., CPU, RAM, and disk requirements) and constraints on location and access rights. Foggy is designed to operate in Fog environments with generally more than three tiers, namely Cloud tier (with high resource capacity), Edge Cloudlets tier (with medium resource capacity), Edge Gateways tier (with low resource capacity), and Swarm of Things tier (IoT devices). In Foggy, resource refers to any computational (such as vCPUs, RAM, and disk), storage or network capacity provided by the nodes of the infrastructure. Foggy uses a set of usage profiles for characterizing the resources. For computational and storage resources, the following profiles are used: General purpose (default profile), Compute optimized, Memory optimized, and Storage optimized. For network resources, the considered profiles are Best Effort (default profile), Interactive application, Signaling and video streaming, Interactive and real-time video. Having focused on performance interference, Shekhar et al. [61] have proposed INDICES (INtelligent Deployment for ubiquitous Cloud and Edge Services) framework which performs online performance monitoring, performance prediction, network performance measurements, and server selection and application migration from the cloud to the fog. The architecture model considered in this work contains a Central Data Center (CDC) connected to a set of Micro Data Centers (MDCs) which are located at the edge. Each MDC comprises a set of computer servers which can be allocated to the CDC for its operations at a specified cost. There are no further discussions on types of resources and their models.

A.2.2 Resource Estimation Models

In order to meet application quality demands, enough resources must be allocated to applications. Accordingly, the required resources for each application should be estimated beforehand for enabling efficient resource management. This is commonly called as resource demand profiling. In this regard, we study the resource estimation techniques employed in the aforementioned works.

The completion time prediction method proposed by Hong et al. [51] utilizes an animation rendering dataset which contains a huge number of records each of which is a rendering job from an animation studio. Each record is described by the resource usage (including CPU usage in percentage and RAM usage in KBs), the characteristics of rendering jobs (e.g., number of frames, number of polygons, and image size in pixels), the network conditions (e.g., the time of sending a job), and the completion time. The capacity planning solution proposed by Noreikis et al. [53] estimates the minimum capacity required for satisfying QoS demands of real-time applications. Their developed profiler measures resource usage while executing a task on a computing node. Based on the measured usage patterns, resource demands are expressed in terms of CPU and GPU utilization (%), network latency (ms), and network bandwidth (kbps). The task execution model used in the HyFog framework [54] characterizes the resource requirements of a task by the required number of CPU cycles. However, they argue that this model can be easily extended to include other resource types. There are no discussions on how to obtain the required number of CPU cycles for a task.

The ENROM framework [55] initializes the applications using a default amount of CPU and memory. However, while the application is running, the proposed auto-scaler mechanism upscales/downscales the allocated resources dynamically. A number of metrics (e.g., round-trip application latency and hardware utilization of CPU and memory) are monitored to make scaling decisions at the auto-scaler component. Therefore, application resource requirements are not estimated beforehand, and the requirements are expressed in terms of application latency (not resource requirements). The ParaDrop framework [56] runs the requested services in virtualized environments called chutes. Resource requirements for chutes are specified in a config file which is necessary for creating chutes. CPU requirement for a chute is specified by a share value which indicates a relative share of the CPU resource that a chute gets compared to what other chutes get. The maximum amount of memory that a chute is allowed to consume is also specified in the config file. It is stated that a strategy based on shares (similar to CPU shares) is planned to be implemented for specifying the network requirements. It is not discussed in the paper how these requirements are extracted.

The resource management framework proposed by Gu et al. [57] considers the overall expected delay (including communication and processing delays) as application quality requirements. Application resource requirements are expressed by storage requirements and processing speed of applications; however, units and resource estimation methods are not discussed. The three-tier edge computing system architecture proposed by Wang et al. [58] expresses application configurations by templates in the form of a text file describing the resource assignments including IP addresses, bandwidth volumes, compute node flavors, security group and etc. There is no discussion on how to determine the resource requirements. In their experimental results, as discussed before, the VM resources are described by the number of vCPUs, size of RAM, and size of storage. Morabito et al. [59] argue that there may be dissimilarity (e.g., in terms of CPU architecture) in different nodes in a heterogeneous environment. Therefore, for each application, different images, where hardware requirements (processor architecture, GPU, and storage) and software requirements (libraries and operating system) are described, must be provided. It is not explained how these requirements are specified.

Deployment requests in Foggy [60] contain the application component to be deployed and a set of optional deployment requirements which are expressed in terms of resource requirements and/or specific application needs such as location and access rights. Foggy employs a set of profiles to express these requirements. The profiles that are used to express application requirements are similar to the ones used to characterize the resources, explained in Section 3.2.1. There is no discussion on how to automatically determine the deployment requirements. Shekhar et al. [61] argue that the performance of an application depends on several factors including:

- the workload: the workload variation can change the performance,
- the hardware hosting platform: application performance can vary from one hardware platform to another in a heterogeneous environment,
- co-located applications that cause performance interference: hypervisors do not provide enough isolation for two reasons, namely presence of non-partitioned shared resources (e.g., cache spaces) and resource overbooking.

In their proposed framework (INDICES) they run applications in a fixed VM configuration (e.g., 2 GB memory, either 1 or 2 VCPUs). However, according to the reasons mentioned above, this fix configuration may lead to various performance levels. They leverage their built performance models to determine whether running an application on a platform causes SLO (Service-Level Objective) violations or not. Therefore, they only consider performance requirements (not resource requirements).

A.2.3 Resource Provisioning Techniques

There is no concrete definition of resource provisioning in the literature. In some works, it is used to describe the whole resource management process, while in some other works, it refers to the resource allocation procedure. In this section, we want to study how the resources are provisioned (i.e., provided) to applications.

The multimedia fog computing platform proposed by Hong et al. [51] utilizes resources from public crowds (e.g., laptops), desktops at the edge of the network, and servers in the cloud to execute animation renderings. Although the available resource dataset they have used to train their models contains resources in VMs, it is not clarified that how the resources are provisioned to users. Noreikis et al. [53] employ Docker containers to provide virtual resources to users in their capacity planning solution. In the HyFog [54] framework, applications tasks can be executed on either mobile devices or cloud servers. Resources in the former case are provided using VMs; however, resource provisioning in devices is not explained. The ENORM framework [55] leverages edge nodes to host servers offloaded from cloud servers. It is argued that edge nodes have limited hardware resources, which makes the containers more appropriate for providing resources to users. LXC containers are used in this framework. ParaDrop WiFi APs [56] are implemented on SBCs whose resources are provisioned in containers (Docker containers in their current implementation) due to their lightweight nature. Gu et al. employ VMs to provision base station resources. The surveillance system architecture proposed by Wang et al. [58] launches a group of VMs in distributed edge cloud servers to provide resources for surveillance tasks. Lightweight characteristics of container-based virtualization are leveraged by Morabito et al. [59] to provision resources in their proposed IoT gateways.

It can be concluded from the studied works that virtualization plays a significant role in resource provisioning, and the virtualization technique for each solution is decided based on the capabilities of employed platforms.

A.2.4 Resource Allocation Strategies

In this part, we study the policies used to allocate resources to applications and map applications on resources. Hong et al. [51] decide where (i.e., which node) to run rendering jobs based on estimated available resources and predicted completion time of jobs on each node. The completion time is predicted using state-of-the-art machine learning algorithms. The details of employed decision-making policies are not discussed. The solution proposed by Noreikis et al. [53] maps long-running and latency insensitive tasks on the cloud and tasks with the shortest tolerable response delay on edge nodes. Additionally, tasks with complementary resource demands are bundled together and mapped on the same node, leading to better resource utilization. They have used the Knapsack algorithm to perform the optimization. In the HyFog framework [54], task offloading decisions are made using a three-layer graph-matching algorithm. The three-layer graph is constructed by taking the offloading space (mobiles, edge nodes, and the cloud) into account. The problem of minimizing the total task execution cost (including the energy cost per CPU cycle and transmission/receiving power costs) is mapped onto the minimum weight-matching problem over the constructed graph, and it is solved using the Edmonds's Blossom algorithm.

The ENORM framework [55] offers several mechanisms for resource management, including handshaking, deployment, auto-scaling, and termination mechanisms. The handshaking is performed between a cloud manager and edge nodes, and it is used to select a node (based on the available free resources on nodes) for application deployment. The auto-scaling mechanism periodically scales the resources allocated to applications whose latency requirements are not met. The termination mechanism terminates an edge service when either it has been idle for a long period or its QoS requirements cannot be satisfied by an edge server deployment. The ParaDrop framework [56] does not provide any resource allocation policies. Rather, the user selects an edge node (i.e., WiFi AP) to deploy its application. Gu et al. [57] investigate QoS guaranteed minimum cost resource management in fog computing supported MCPS. They formulate the cost minimization problem in a form of mixed-integer nonlinear programming (MINLP), and they linearize it as mixed-integer linear programming (MILP) problem to cope with the high complexity of solving MINLP. Further more, they propose a low-complexity two-phase LP-based heuristic algorithm to solve the MILP problem. In their problem formulation they consider four constraints, namely 1) user association constraints (each user must be associated with a base station, and a subcarrier in the BS must be allocated to the user), 2) task distribution constraints (the application data uploaded to a BS can be distributed to other BSs to get processed), 3) VM placement constraints (VMs must be deployed on BSs, and their resource requirements must not exceed the capacity of BSs), and 4) QoS constraints (the overall expected delay, including communication and processing delay, shall not exceed the application delay constraint). The total cost they seek to minimize includes the total VM deployment cost, uploading cost, and inter-BS communication cost.

Wang et al. [58] offer an elastic resource allocation mechanism in their surveillance system where computing resources are reallocated when emergency events happen. To do so, at any point in time, the closest edge nodes to the tracked object are selected to run surveillance tasks. When resource shortage happens, a part of the workload is transferred to another node whose selection depends on its distance to the monitor that captures the object's video. It can be implied that their approach minimizes the deployment costs by minimizing communication latencies. In the Foggy framework [60], to map applications to edge nodes, the orchestrator filters the nodes that can satisfy application requirements. Then, it sorts the filtered nodes according to a priority function whose details are not discussed. Subsequently, the node with the highest rank will be chosen to deploy the application. The objective of the INDICES framework [61] is to assure the SLOs (i.e., response times) for all the applications (by identifying SLO violations and migrating applications from the cloud to edge servers) while minimizing the overall deployment cost. To identify the SLO violations, application execution times are estimated using performance and interference profiles. An interference profile of an application identifies the degree to which that application will degrade the performance of other running applications on the host—called pressure—and how much its own performance will degrade due to interference from other applications—called sensitivity. Accordingly, the framework offers a performance interference-aware server selection algorithm where the SLO-violated applications are migrated to the edge nodes in such a way that the so-called pressure and sensitivity do not cause SLO violations, and furthermore, the overall deployment cost is minimized. The optimization problem is solved using a heuristic-based algorithm since the problem is an NP-Hard one.

A.2.5 Resource Management Architectures

In this section, we study the architecture introduced by the prior works to perform resource management. The framework proposed by Hong et al. [51] has three components, namely an available resource predictor, a completion time predictor, and a job scheduler. The job scheduler decides where to deploy a job based on the information provided by the resource predictor and completion time predictor. The ENORM framework [55] works across three tiers, namely the cloud tier, the edge node tier, and the user device tier. The cloud tier is where the application servers are located, and a cloud server manager runs on each application server. A cloud server manager sends requests to edge nodes, deploys services on edge nodes, and updates the global view of the application server based on the deployments. Each edge node has several components to receive requests from the cloud server manager, negotiate with it, deploy applications upon accepted requests, monitor resources and applications, and perform the auto-scaling mechanism.

The ParaDrop framework [56] has two main resource management agents, namely the ParaDrop backend and the ParaDrop daemon. The ParaDrop backend manages all the resources of the platform in a centralized manner and provides APIs for users to deploy services on the gateways. The ParaDrop daemon runs on each Access Point to perform all the functions required by the ParaDrop platform, including registering the AP to the backend, monitoring the status of AP and reporting to the backend, resource and process management, and receiving RPCs (Remote Procedure Calls) and messages from the backend and performing lifecycle management of chutes (i.e., application containers) accordingly. The architecture proposed by Wang et al. [58] consists of three tiers, namely applications tier, edge computing tier, and data tier. The applications tier contains resource requirements of tasks, plans resource allocations and configurations, and monitors the running status of the applications. The edge computing tier contains an edge control node which performs resource orchestration (to satisfy resource requirements of tasks) and a SDN controller which monitors, configures, and manages VMs. The data tier contains terminal monitors that collect and upload real-time video data to the nearest available compute node.

The architecture proposed by Morabito et al. [59] contains an IoT Application Orchestrator which determines which software (i.e., application) is used for processing the data of a specific device as well as the best location (data center or gateway) for deploying it. It is stated that the orchestrator takes the hardware requirements (processor architecture, GPU, storage) and software requirements (libraries, operating system) of processing software into account during its decision makings. Foggy [60] is an architectural framework which offers the functionality of negotiation, scheduling, and workload placement. The management architecture is composed of an inventory, a negotiator, and an orchestrator. The inventory maintains the status of the infrastructure (i.e., available resources and their location). The negotiator decides whether to accept or reject deployment requests based on the status of the infrastructure. For the accepted deployment requests, the orchestrator deploys application components on the node that best satisfies the deployment requirements.

The architecture model considered in the INDICES framework [61] contains a Central Data Center (CDC) connected to a set of Micro Data Centers (MDCs) which are located at the edge. Each MDC comprises a set of computer servers which can be allocated to the CDC for its operations at a specified cost. A global manager on the CDC is responsible for detecting and mitigating global SLO violations. On each MDC, one of its servers acts as local manager which is responsible for data collection, performance estimation, latency measurements, and MDC-level decision making. During run-time, the global manager identifies the SLO violations, and the local managers decide where to migrate the SLO-violated applications. The works that are not discussed in this section have not made a clear discussion about their architecture.

A.2.6 Summary and Conclusions

The reviewed techniques are summarized in Table 4.

Table 4. Summary of reviewed resource management works.

Reference	Managed Resources	Resource Demand Profiling	Resource Provisioning	Resource Allocation	Architecture
Hong et al. [51]	CPU, Memory, Disk, Network	Machine learning algorithms are trained using an animation rendering dataset to predict resource demands.	VMs (not explicitly stated)	VM placement is decided according to estimated available resources and predicted completion time of jobs. Machine learning algorithms are used to predict the completion time.	A hybrid architecture comprised of an available resource predictor, a completion time predictor, and a job scheduler.
Noreikis et al. [53]	CPU, GPU, Network	A profiler extracts resource usage patterns while applications are running. Demands are expressed in CPU/GPU utilization, network latency, and network bandwidth.	Docker containers	Long-running and delay insensitive tasks run in the cloud. Delay sensitive applications run on edge nodes. Applications with complementary requirements run on the same nodes. Knapsack algorithm is used.	Not explained.
Chen et al. [54]	CPU,Network	Resource demands are expressed in the required number of CPU cycles. Profiling approach is not discussed.	VMs in cloud servers; provisioning for edge devices is not discussed.	Task offloading decisions are made using a three-layer graph-matching algorithm. The total task execution cost is minimized. Edmonds's Blossom algorithm is used.	Not explained.
Wang et al. [55]	CPU, Memory	Required resources are not estimated beforehand. Latency requirements are used to decide about resource scalings.	LXC containers	Container placement is decided according to available free resources. Scaling decisions are made based on latency violations. Termination mechanism is proposed as well.	A hybrid architecture comprised of a central cloud manager and distributed managers on edge nodes.
Liu et al. [56]	CPU, Memory, Network	Resource demands are expressed by CPU shares and the maximum memory usage. Profiling approach is not discussed.	Docker containers	Performed by users.	A hybrid architecture consisting of a centralized backed in the cloud and distributed daemons on edge nodes.
Gu et al. [57]	CPU	Resource demands are expressed by storage requirements and required processing speed.	VMs	The total cost (VM deployment cost, uploading cost, and communication cost) is minimized subject to resource and quality constraints. The problem is formulated as a MINLP problem and solved using a LP-based heuristic algorithm.	Not explained.
Wang et al. [58]	CPU, Memory, Disk	Demands are described by templates containing the resource assignments including IP addresses, bandwidth volumes, compute node flavors, security group and etc.	VMs	Reallocation decisions are made with the goal of minimizing communication latencies.	A three-tier architecture where system managers are located at the application tier, and edge node control and SDN controller are located at the edge computing tier.
Morabito et al. [59]	CPU, GPU, Disk	Hardware requirements (processor architecture, GPU, and storage) and software requirements (libraries and operating system) are described in application images. Profiling approach is not discussed.	Containers	Container placement is decided according to required hardware and software resources. Policies are not discussed.	A centralized management architecture containing an IoT orchestrator.
Santoro et al. [60]	CPU, Memory, Disk, Network	Deployment requirements are expressed using a set of profiles in terms of resource requirements and/or specific application needs such as location and access rights. Profiling approach is not discussed.	Docker containers	A priority function is used to sort the edge nodes with sufficient resources. Containers are placed on the nodes with the highest ranks.	The management architecture is composed of an inventory, a negotiator, and an orchestrator.
Shekhar et al. [61]	Not discussed	Only performance requirements are considered.	Containers inside VMs	A performance interference-aware server selection method is used to migrate SLO-violated applications to edge nodes. The optimization goal is to minimize the SLO violations and the overall deployment cost. The problem is solved using a heuristic-based algorithm.	A hybrid architecture composed of a centralized global manager on the central data center and distributed local managers on edge micro data centers.

A.3 References

[1] M. Portnoy, Virtualization essentials, vol. 19. John Wiley & Sons, 2012.

[2] J. Sahoo, S. Mohapatra, and R. Lath, “Virtualization: A survey on concepts, taxonomy and associated security issues,” in Proc. 2nd Intl. Conf. on Computer and Network Technology, pp. 222–226, IEEE, 2010.

[3] B. Jennings and R. Stadler, “Resource management in clouds: Survey and research challenges,” Journal of Network and Systems Management, vol. 23, pp. 567–619, Jul 2015.

[4] C. A. Waldspurger, “Memory resource management in VMware ESX server,” ACM SIGOPS Operating Systems Review, vol. 36, no. SI, pp. 181–194, 2002.

[5] A. Velte and T. Velte, “Microsoft virtualization with Hyper-V,” McGraw-Hill, Inc., 2009.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in ACM SIGOPS operating systems review, vol. 37, pp. 164–177, ACM, 2003.

[7] A. Nelson, A. B. Nejad, A. Molnos, M. Koedam, and K. Goossens, “Comik: A predictable and cycle-accurately composable real-time microkernel,” in Proceedings of the conference on Design, Automation & Test in Europe, p. 222, European Design and Automation Association, 2014.

[8] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, “Xtratum: a hypervisor for safety critical embedded systems,” in Proc. 11th Real-Time Linux Workshop, pp. 263–272, Citeseer, 2009.

[9] R. Kaiser and S. Wagner, “The PikeOS concept: History and design,” SysGO AG White Paper. Available: http://www.sysgo.com, 2007.

[10] D. Marshall, “Understanding full virtualization, para-virtualization, and hardware assist,” VMware White Paper, p. 17, 2007.

[11] “Virtualbox.” https://www.virtualbox.org. Accessed: 2019-02-03.

[12] “Parallels desktop for Mac.” https://www.parallels.com/products/desktop. Accessed: 2019-02-03.

[13] F. Bellard, “Qemu: a fast and portable dynamic translator,” in USENIX Annual Technical Conference, FREENIX Track, vol. 41, p. 46, 2005.

[14] I. Habib, “Virtualization with KVM,” Linux Journal, vol. 2008, no. 166, p. 8, 2008.

[15] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library operating systems for the cloud,” in Proc. 18th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’13), pp. 461–472, ACM, 2013.

[16] D. R. Engler, M. F. Kaashoek, et al., “Exokernel: An operating system architecture for application-level resource management,” vol. 29. ACM, 1995.

[17] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, “The design and implementation of an operating system to support

distributed multimedia applications," IEEE Journal on Selected areas in communications, vol. 14, no. 7, pp. 1280–1297, 1996.

[18] M. J. De Lucia, "A survey on security isolation of virtualization, containers, and unikernels," tech. rep., US Army Research Laboratory Aberdeen Proving Ground United States, 2017.

[19] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," IEEE Network, vol. 32, pp. 102–111, Jan 2018.

[20] "Haskell lightweight virtual machine (halvm)." https://galois.com/project/halvm. Accessed: 2019-02-03.

[21] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarEl, D. Marti, and V. Zolotarov, "Optimizing the operating system for virtual machines," in USENIX Annual Technical Conference (USENIX ATC '14), pp. 61–72, 2014.

[22] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in 7th Intl. Conf. on Cloud Computing Technology and Science (CloudCom), pp. 250–257, IEEE, 2015.

[23] J. Martins, M. Ahmed, C. Raiciu, and F. Huici, "Enabling fast, dynamic network processing with ClickOS," in Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, pp. 67–72, ACM, 2013.

[24] C. Hong and B. Varghese, "Resource management in fog/edge computing: A survey," CoRR, vol. abs/1810.00305, 2018.

[25] M. Helsley, "LXC: Linux container tools," IBM developerWorks Technical Library, vol. 11, 2009.

[26] "Lxd." https://linuxcontainers.org/lxd/introduction. Accessed: 2019-02-03.

[27] "Containers on windows." https://docs.microsoft.com/en-us/virtualization/windowscontainers/ab. Accessed: 2019-02-03.

[28] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," Linux J., vol. 2014, Mar. 2014.

[29] "Openvz: Open source container-based virtualization for Linux." https://openvz.org. Accessed: 2019-02-03.

[30] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in Proc. 2nd Intl. SANE Conference, vol. 43, p. 116, 2000.

[31] D. Price and A. Tucker, "Solaris zones: Operating system support for consolidating commercial workloads," in LISA, vol. 4, pp. 241–254, 2004.

[32] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in Proc. IEEE Intl. Conf. on Cloud Engineering, pp. 386–393, IEEE, 2015.

[33] J. Hwang, S. Zeng, F. y Wu, and T. Wood, "A component-based performance comparison of four hypervisors," in Proc. IFIP/IEEE Intl. Symp. on Integrated Network Management (IM 2013), pp. 269–276, IEEE, 2013.

[34] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, “Containers and virtual machines at scale: A comparative study,” in Proc. 17th Intl. Middleware Conference (Middleware ’16), pp. 1:1–1:13, ACM, 2016.

[35] A. Binu and G. S. Kumar, “Virtualization techniques: a methodical review of Xen and KVM,” in Proc. Intl. Conf. on Advances in Computing and Communications, pp. 399–410, Springer, 2011.

[36] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in Proc. IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS), pp. 171–172, IEEE, 2015.

[37] A. Babu, M. Hareesh, J. P. Martin, S. Cherian, and Y. Sastri, “System performance evaluation of para virtualization, container virtualization, and full virtualization using Xen, OpenVZ, and Xenserver,” in Proc. 4th Intl. Conf. on Advances in Computing and Communications, pp. 247–250, IEEE, 2014.

[38] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, “Analysis of virtualization technologies for high performance computing environments,” in Proc. 4th IEEE Intl. Conf. on Cloud Computing, pp. 9–16, IEEE, 2011.

[39] K. Adams and O. Agesen, “A comparison of software and hardware techniques for x86 virtualization,” ACM SIGOPS Operating Systems Review, vol. 40, no. 5, pp. 2–13, 2006.

[40] F. Huici, F. Manco, J. Mendes, and S. Kuenzer, “VMs, unikernels and containers: Experiences on the performance of virtualization technologies,”

[41] N. G. Bachiega, P. S. Souza, S. M. Bruschi, and S. d. R. de Souza, “Container-based performance evaluation: A survey and challenges,” in Proc. IEEE Intl. Conf. on on Cloud Engineering (IC2E), pp. 398–403, IEEE, 2018.

[42] M. Satyanarayanan, “The emergence of edge computing,” Computer, vol. 50, pp. 30–39, Jan 2017.

[43] “What is edge?” https://www.etsi.org/newsroom/blogs/entry/what-is-edge. Accessed: 2019-02-03.

[44] A. V. Dastjerdi and R. Buyya, “Fog computing: Helping the internet of things realize its potential,” Computer, vol. 49, pp. 112–116, Aug 2016.

[45] S. S. Manvi and G. K. Shyam, “Resource management for infrastructure as a service (IaaS) in cloud computing: A survey,” Journal of Network and Computer Applications, vol. 41, pp. 424– 440, 2014.

[46] P.-O. Östberg, J. Byrne, P. Casari, P. Eardley, A. F. Anta, J. Forsman, J. Kennedy, T. Le Duc, M. N. Marino, R. Loomba, et al., “Reliable capacity provisioning for distributed cloud/edge/fog computing applications,” in Proc. European Conf. on Networks and Communications (EuCNC), pp. 1–6, IEEE, 2017.

[47] J. Hamilton, “Cooperative expendable micro-slice servers (cems): low cost, low power servers for internet-scale services,” in Proc. Conf. on Innovative Data Systems Research (CIDR09), Citeseer, 2009.

[48] A. Hameed, A. Khoshkbarforoushha, R. Ranjan, P. P. Jayaraman, J. Kolodziej, P. Balaji, S. Zeadally, Q. M. Malluhi, N. Tziritas, A. Vishnu, et al., “A survey and taxonomy

on energy efficient resource allocation techniques for cloud computing systems," Computing, vol. 98, no. 7, pp. 751–774, 2016.

[49] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox, and J. Singer, "Commodity single board computer clusters and their applications," Future Generation Computer Systems, vol. 89, pp. 201–212, 2018.

[50] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over RaspberryPi," in Proc. 18th Intl. Conf. on Distributed Computing and Networking, p. 16, ACM, 2017.

[51] H.-J. Hong, J.-C. Chuang, and C.-H. Hsu, "Animation rendering on multimedia fog computing platforms," in IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom), pp. 336–343, IEEE, 2016.

[52] S. Shen, V. v. Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud data centers," in Proc. 15th IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Computing, pp. 465–474, May 2015.

[53] M. Noreikis, Y. Xiao, and A. Yl-Jaiski, "Qos-oriented capacity planning for edge computing," in Proc. IEEE Intl. Conf. on Communications (ICC), pp. 1–6, May 2017.

[54] X. Chen and J. Zhang, "When d2d meets cloud: Hybrid mobile task offloadings in fog computing," in Proc. IEEE Intl. Conf. on Communications (ICC), pp. 1–6, May 2017.

[55] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos, "Enorm: A framework for edge node resource management," IEEE Transactions on Service Computing, pp. 1–1, 2018.

[56] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the networks extreme edge," in Proc. IEEE/ACM Symp. on Edge Computing (SEC), pp. 1–13, Oct 2016.

[57] L. Gu, D. Zeng, S. Guo, A. Barnawi, and Y. Xiang, "Cost efficient resource management in fog computing supported medical cyber-physical system," IEEE Transactions on Emerging Topics in Computing, vol. 5, pp. 108–119, Jan 2017.

[58] J. Wang, J. Pan, and F. Esposito, "Elastic urban video surveillance system using edge computing," in Proc. Workshop on Smart Internet of Things (SmartIoT '17), pp. 7:1–7:6, ACM, 2017.

[59] R. Morabito and N. Beijar, "Enabling data processing at the network edge through lightweight virtualization technologies," in Proc. Intl. Conf. on Sensing, Communication and Networking (SECON Workshops), pp. 1–6, June 2016.

[60] D. Santoro, D. Zozin, D. Pizzolli, F. D. Pellegrini, and S. Cretti, "Foggy: A platform for workload orchestration in a fog computing environment," in IEEE Intl. Conf. on Cloud Computing Technology and Science (CloudCom), pp. 231–234, Dec 2017.

[61] S. Shekhar, A. D. Chhokra, A. Bhattacharjee, G. Aupy, and A. Gokhale, "Indices: exploiting edge resources for performance-aware cloud-hosted services," in Proc. 1st IEEE Intl. Conf. on Fog and Edge Computing (ICFEC), pp. 75–80, IEEE, 2017.

[61] Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, van Kampenhout, Reinier, Rasool Tavakoli,

Juan Valencia, Ahmadi Balef, Hadi, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi. "NOC-Based Multi-Processor Architecture for Mixed Time-Criticality Applications", In Handbook of Hardware/Software Codesign, Soonhoi Ha and Jurgen Teich (editors), Springer, 2017.