Real-Time FaaS: Towards a Latency Bounded Serverless Cloud

Márk Szalay*‡, Péter Mátray†, László Toka*‡,
*Budapest University of Technology and Economics, Hungary,
†Ericsson Research, Hungary ‡MTA-BME Network Softwarization Research Group, Hungary

Abstract—Today, Function-as-a-Service is the most promising concept of serverless cloud computing. It makes possible for developers to focus on application development without any system management effort: FaaS ensures resource allocation, fast response time, schedulability, scalability, resiliency, and upgradability. Applications of 5G, IoT, and Industry 4.0 raise the idea to open cloud-edge computing infrastructures for time-critical applications too, i.e., there is a strong desire to pose real-time requirements for computing systems like FaaS. However, multi-node systems make real-time scheduling significantly complex since guaranteeing real-time task execution and communication is challenging even on one computing node with multi-core processors. In this paper, we present an analytical model and a heuristic partitioning scheduling algorithm suitable for real-time FaaS platforms of multi-node clusters. We show that our task scheduling heuristics could outperform existing algorithms by 55%. Furthermore, we propose three conceptual designs to enable the necessary real-time communications. We present the architecture of the envisioned real-time FaaS platform, emphasize its benefits and the requirements for the underlying network and nodes, and survey the related work that could meet these demands.

Index Terms—real-time scheduling, real-time cloud, real-time FaaS, partitioned scheduling, partitioned-EDF, heterogeneous multiprocessor, multi-node and multiprocessor system.

1 Introduction

With the advent of 5G, IoT, and Industry 4.0, the idea of running delay-sensitive applications in the cloud instead of special-purpose hardware is gradually gaining ground. As [1] argues, cyber-physical systems, IoT, and cloud computing together create Industry 4.0, i.e., makes the concept of "smart factory" alive. This paves the way for use-cases that were considered impossible before, such as remote-controlled factories. Novel applications of Industry 4.0 require reliable wireless connections, ultra-low end-to-end (E2E) latency, high data rates, and supporting a massive number of devices. Fortunately, these characteristics are enabled by 5G [2], [3]. E.g., the authors of [4] present how 5G can facilitate a distributed robotics control system. They allow time-critical and computationally exhaustive operations to be offloaded to the cloud using 5G URLLC (Ultra-reliable low-latency communication) between the cloud and the robot. These technologies lead to the idea of opening up cloud and edge computing infrastructures for time-critical applications, such as cloud-assisted safety systems, tactile internet, virtual and augmented reality applications.

Thanks to its flexibility benefits, serverless computing has become one of the most popular paradigms in the field of cloud computing. It allows developers to focus only on application development and not care about the server management and auxiliary functions by providing resource allocation, fast response time, schedulability, scalability, resiliency, and upgradability of the deployed applications. Function-as-a-Service (FaaS) – the best-known implementation of serverless computing – makes it possible for users to run their microservice applications without any management effort. Most of the FaaS platforms rely on container technologies, like LXC or Docker, and container orchestration systems such as Kubernetes. Typically, the users' source code is wrapped into containers that are scheduled and deployed by Kubernetes. Containerization provides isolation of processes and libraries with low-performance overhead.

Several research initiatives have tackled the idea of running real-time (RT) applications in containers [5], [6], [7], but it still seems to be an undiscovered research field today. This is especially true if both RT and traditional containers are scheduled on the same hardware. Mixed-Criticality Systems (MCS) are able to handle such apps with various levels of criticality. They ensure the correct execution of critical tasks while sharing the underlying hardware resources. To our knowledge, currently, there is no solution that would be capable of scheduling RT tasks on a server cluster. No wonder, since such a system has many demands which are still actively researched: 1) underlying networks need to be deterministic, 2) the operating systems (OS) of the servers must operate in real-time, 3) it need to support spatial and temporal real-time scheduling of tasks, 4) it has to take into account the heterogeneity of the computing infrastructure.

In this paper, we introduce – to the best of our knowledge the first and only – real-time FaaS (RT-FaaS) system, which extends the current RT platforms by bringing in the distributed aspect. We collect the related technical requirements and research challenges essential for such system. In addition, we propose possible methods for both the RT communication and the real-time scheduling of the functions to ensure the RT execution. Besides, by summarizing the real-time results published in different computer science fields, we aim to start a new research direction, called real-time FaaS, and encourage the research community to contribute.

The structure of the paper is the following. In Sec. 2 we define the RT-FaaS, its the two major challenges and the feature requirements. Sec. 3 summarizes the related works.

Sec. 4 proposes three conceptual designs to enable the RT communication of the deployed functions of an RT-FaaS. Sec. 5 elaborates on the problem of RT function scheduling and introduces the assumed environment models. We argue for partitioning scheduling in systems like RT-FaaS; Sec. 6 presents our proposed partition algorithm. We evaluate the algorithm in Sec. 7. Finally, Sec. 8 concludes the paper.

A preliminary version of this paper appears as a conference paper [8]. As novel content, (1) we define the RT-FaaS system, identify its two major challenges, and determine the requirements that such a system should meet, (2) we propose three conceptual designs to ensure RT communications in an RT-FaaS platform, (3) we add corner cases when the proposed partitioning algorithm – related to RT partitioned scheduling approach – returns optimal partitioning, (4) we present, in the worst-case, how much the result of proposed algorithm deviates from the optimum, and finally (5) we have extended the related work with additional relevant research works from this topic.

2 The concept of real-time FaaS

In the following, we argue why the FaaS serverless systems are suitable to run users' RT applications in the cloud. We define what the RT-FaaS cloud system is, what the major challenges are to tackle, and the feature requirements posed against the FaaS infrastructure to handle critical and non-critical functions jointly.

FaaS as a target platform for RT applications. The authors of [9] mention in their paper, many students at UC Berkeley complain why there is not a "cloud button" that could be theoretically pushed to run users' simple program code in an optimized way within the cloud. They are not the only ones who still find the existing cloud approaches too complex and challenging to use: nowadays, many cloud platforms exist offering different types of virtualized instances to run user applications with totally different pricing methods. This means that it is difficult to i) choose among the available cloud providers, ii) configure the used virtualized system, and iii) optimize it to get efficient application performance for a minimized cost. Plus, these tasks become more challenging if users want to run RT applications, i.e., apps which are guaranteed to return before a predefined deadline.

We argue that a *FaaS serverless computing system is the most suitable solution to run RT and non-RT applications* both from the cloud providers' and the users' point of view. On the one hand, it is not easy to build cloud-ready applications due to the available wide range of cloud providers, services, and technologies. Furthermore, the developers or cloud users, have to deal with the kernel, scheduler, virtual machine (VM) or network configuration, and the parallel execution of the application tasks to get the required performance of their applications. Fortunately, FaaS provides the *highest level of abstraction to run our application in the cloud*: users should define only a single-threaded source code, install it into the FaaS system, and pay only for the execution.

On the other hand, FaaS should also be preferred by the cloud providers, since this way *they have the greatest possible control and service configuration over their infrastructure*. Moreover, offering only a high-level interface to deploy user functions prevents their physical infrastructure from being misconfigured by users and thus being unable to provide the required QoS. E.g., in the case of RT functions, the cloud provider would uselessly provide a VM that would be able to run the RT application if the user uses a wrong operating system scheduler with an incorrect configuration. In such a case, even though the cloud provider ensures the necessary infrastructure environment, the user's RT applications would still not be able to meet their deadlines.

RT-FaaS serverless computing system.

We define the RT-FaaS cloud system in Def. 1. Multiple nodes, including numerous CPUs, form a server cluster on which the specified RT-FaaS platform runs. The users can demand their functions to deploy, and depending on whether it is an RT function or not, they provide maximum tolerated *response time* and the function's *period*. The former determines the maximum time for the function's response to arrive back to the user. The latter defines the trigger frequency of the function. A practical example for such an RT function might be a video frame processing tool [10] that receives each snapshot image of a video stream, analyzes it, and returns the detections to the user. E.g., in the case of an 80 *fps* video stream, a new frame arrives every 12 *ms*, which is the period of the video processing function.

Definition 1. ***RT-FaaS cloud system:*** *RT-FaaS is the infrastructure, including computing servers and the network connecting them, on which the complete FaaS software stack runs to offer cloud computing resources for users' time critical (RT) or non-critical (non-RT) functions such that the worst-case response times of the deployed RT functions are guaranteed to be below the predefined value.*

FaaS systems, by definition, exist to run event-driven and stateless functions. Users install their functions in advance, and they are invoked repeatedly whenever a new input or event arrives. For an event, the FaaS creates a worker which executes the user function within a container. The number of worker containers is scaled elastically according to the incoming events. One major limitation of current FaaS systems is that the timing of the worker invocations is unpredictable.

In the case of RT functions, we must rethink the way how functions are executed. When users install their RT functions, they also provide the invocation period (i.e., how frequently the incoming input data to process arrives) and the response time within which it must return. We assume periodic functions as RT tasks that process the periodically incoming data. The scheduler of the RT-FaaS system determines the CPU of a server within the server cluster, which will run the worker (i.e., the container) that executes the RT functions. Due to the real-time requirements, we must avoid cold executions. This results in that they should be running on predetermined CPUs and processing the incoming data in each invocation period. Thus, it can be provided that the output data returns before the required deadline, and the function remains warm. Since we assume periodic functions, they run as long as the incoming input data arrives according to the predefined period.

One might ask how the dynamic scaling of RT FaaS applications is supported. In this work, we assume *periodic*

RT functions to run on the proposed RT-FaaS platform. This means, they all have their invocation period and, by definition, they cannot be triggered more frequently than this period. Consequently, only *one function invocation per period* is allowed. If the workload to process increases, more than one RT function should be run to process the input. In that case, the RT-FaaS users should deploy new RT instances of the already deployed RT application with the same invocation period. Considering *aperiodic* or *sporadic* RT functions where the one function invocation per period restriction is insufficient, is out of the scope of this paper.

RT-FaaS challenges. The goal of the RT-FaaS is to make it possible for users to externalize their applications into the RT-FaaS cloud even if it requires deterministic execution. To accomplish this, the two major challenges below need to be tackled.

Definition 2. ***Two major challenges of the RT-FaaS:***

1) *Real-time execution of user demanded functions*
2) *Real-time communication within the RT-FaaS system*

The *RT execution* of functions ensures, if the input data to process is available, the RT-functions are going to be executed and their output will be generated before the function's deadline, thus providing the response to the user in time. To this end, the RT-FaaS must guarantee both *spatial* and *temporal scheduling* of the incoming user demanded functions, i.e., it determines where – on which node's CPU – and when – in which time interval – to run functions to meet their deadline requirements if they have any.

The *real-time communication* of the deployed functions is also essential. Although real-time execution ensures the function generates its return value before the deadline, it does not mean it will also be transmitted back to the user consistently in time. The underlying network, both the physical one among nodes and the virtual one within the nodes, need to be deterministic to provide the worst-case transmission time between two components of the managed RT-FaaS cloud computing system. *All in all, both real-time execution and communication of the functions are required for an RT-FaaS. They jointly provide the i) function invocation and input data transmission, ii) the function execution – thus generating the required output – the iii) cloud service invocation (e.g., DB writing) if it is necessary and iv) the return value transmission back to the user in time before the user-defined deadline.*

RT-FaaS requirements. We argue the RT-FaaS must meet several feature requirements to be able to run RT and non-RT functions together such that deadline requirements are met, while many of the well-known benefits of cloud systems are still ensured. We defined these requirements in Def. 3.

Definition 3. ***Requirements for RT-FaaS:***

1) *Ensuring the isolation of functions (multitenancy)*
2) *Allowing critical and non-critical functions to be deployed (mixed criticality system)*
3) *Offering RT and non-RT cloud services to the tenants (third party cloud services)*
4) *Running on a multi-node system*
5) *Supporting heterogeneous hardware devices*
6) *Ensuring end-to-end (E2E) bounded latency between the user and the worker where RT-functions run.*
7) *Ensuring function to RT cloud service bounded latency*

The tenants' functions must be isolated from each other and from the underlying host operating system (OS) to protect the shared infrastructure and ensure the tenants' privacy (Req. 1). E.g., containers in different namespaces in which user functions run could fulfill this requirement. We argue that RT-FaaS must support both RT and non-RT functions (Req. 2), prioritizing the execution of the former above the latter. This ensures the necessary RT finishing time for the critical functions. Furthermore, the RT-FaaS could offer RT and non-RT services to the users, which can be used during the deployed function's operation (Req. 3). Such a service could be a low-latency RT cloud database (DB), event streaming and messaging tool, or an artificial intelligence model. Examining, how RT services could be enabled in computation systems is out of the scope of this paper, however, it is an actively researched topic [11], [12], [13]. The RT-FaaS should manage not a single, but multiple nodes, i.e., it could operate on a server cluster (Req. 4). Furthermore, these servers might be heterogeneous nodes containing different types and amounts of CPUs, memory modules, Network Interface Controllers (NIC), etc. (Req. 5). The RT-FaaS system includes single or multiple ingress points, through which the users could communicate their deployed RT or non-RT functions. We argue the network between the ingress points and the functions needs to be deterministic (Req. 6), thus the worst-case response time of the functions could be guaranteed.

RT-FaaS goal.

Several optimization goals can be defined for such an RT-FaaS system. In this paper, we address the question of *how to run RT functions in the most cost-effective way to maximize the economic advantage of cloud shared resources for non-RT ones*? In Sec. 3, we list the prior arts which are essential for an RT-FaaS system to provide the determinism of the data transmission time and function execution. However, these are complementary building blocks, not complete solutions. We are not aware of any system which combines the flexibility of a FaaS system with real-time guarantees.

3 Related Work

We summarize the current state of the art of meeting the challenges and requirements of RT-FaaS in this section.

Deterministic Physical Networks: The underlying network of the RT-FaaS system should be deterministic. Such networks are highly reliable, providing bounded latency and low jitter. IEEE 802.1 TSN (Time-Sensitive Networking) [14] and IETF DetNet [15] are two known approaches which aim to provide deterministic and reliable packet forwarding for network services. [16] analyzes the similarities and differences and give a survey of the published standards and possible future areas. [17] surveys the studies of TSN and DetNet that specifically target the support of Ultra-Low Latency in 5G networks.

Deterministic Virtual Networks: However, not only the network must be latency bounded, but other communications within the servers as well. Gutiérrez et al. present a study [18] about the Linux communication stack meant for real-time robotic applications. They proved, under appropriate configuration, the Linux kernel greatly enhances

the deterministic nature of communications if there is no significant non-critical concurrent traffic. On the other hand, there exists network stacks with the specific purpose of providing deterministic networking. Such an approach is RTnet [19], an open-source hard real-time network protocol stack for Linux equipped with real-time extension RTAI or Xenomai. RTnet implements UDP/IP, TCP/IP, ICMP, and ARP in a deterministic way, and it is able to handle jointly RT and non-RT traffic.

Unfortunately, the Linux network stack nowadays is still unsuitable for time-critical network applications [20]. However, instead of applying determinism, another approach is to bypass the kernel and retrieve data from the NIC directly to the app. Several solutions exist for this [21], e.g. DPDK.

Jesus Sanchez-Palencia from Intel, at Embedded Linux Conference [22] presented how they enable TSN in the upstream direction within a Linux node by using TSN supported NIC, assigning traffic priorities to its hardware queues, and steering the traffic from the application into the correct Tx queue. In the presentation he did not touch on how to deal with the dynamic memory allocation, which causes unpredictable delays in Linux kernel. It was mentioned, socket AF_XDP (Address Family eXpress Data Path) could mitigate this. One year later, also at the Embedded Linux Conference, Björn Töpel and Magnus Karlsson [23] presented the AF_XDP sockets that have been designed from the ground up to be able to deterministically deliver sub-microsecond packet latencies and process millions of packets per second. AF_XDP allows zero-copy i.e., network packets are not copied between kernel and user space memories, instead accessed directly by applications.

Li et al. in [24] realized the Xen hypervisor handles RT and non-RT traffic through its common interface and identified limitations that could result in long or unpredictable network traffic latencies between virtual machines. They propose a Virtualization-Aware Traffic Control framework to improve predictability and to reduce the delay for critical applications.

Real-time containers: Struhár et al. provide a survey [5] summarizes the research community's effort on real-time properties in container-based virtualization. [6] provides a reference architecture for implementing the RT container concept on top of a Linux kernel patched with a hard real-time co-kernel. Abeni et al. [7] propose to use a real-time deadline-based scheduling policy built into the kernel to provide temporal scheduling guarantees to different co-located containers.

Real-time CPU scheduling:

Suppose physical and virtual networks, and containers themselves all provide real-time services. In that case, the remaining essential criteria is to make the hosts' operating system (OS) real-time too, i.e., apply RT CPU scheduling. When only a single CPU exists, the basic scheduling solution is Earliest Deadline First (EDF) [25]. If multiple CPU exist, the multiprocessor scheduling approaches can be divided into two categories: *partitioned* and *global* techniques. The former uses a partitioning method to spread the tasks onto CPUs and run a mono-processor scheduling algorithm (e.g., EDF) on each core. This is a static scheduling approach, i.e., the task migration between the cores is not allowed. On the other hand, global scheduling is a dynamic approach that has a single scheduler that determines when and on which CPU to run the task. Here task migration is allowed, i.e., each task could run on each processor at any time.

Several results have been published for partitioned scheduling on multicore systems. However, these are generally tailored for a single node environment, including multiple CPUs. In contrast, we believe for RT-FaaS, multiple CPUs of multiple nodes are to be managed to execute both RT and non-RT tasks. The heterogeneity of available CPUs also must be taken into account, which can greatly affect the quality of scheduling. The processors of CPU pools are typically divided into three categories: *identical*, *uniform heterogeneous*, and *unrelated heterogeneous* CPUs. In identical multiprocessor platforms, all CPU processing speeds are the same. The uniform heterogeneous multiprocessor platforms contain CPUs with various processing speeds. Finally, in unrelated heterogeneous multiprocessor platforms, the processing speed depends on both the CPU speed and the function being executed. E.g., a GPU would execute graphics jobs faster than CPUs.

Existing papers which study the partitioned scheduling problem of RT tasks typically consider either identical cores or CPUs as [26], [27], [28] or only a limited heterogeneity among of them (e.g., dual-core types of the ARM big.LITTLE architecture) [29], [30], [31]. [32], [33] propose methods running in non-polynomial time to solve the partitioning problem. Baruah in [34] presents the Heterogeneous Multiprocessor Partitioning Problem and his approximation algorithm that aims to minimize the maximum fraction of the capacity of any CPU that is utilized to process RT tasks.

Global scheduling is also a highly investigated topic. E.g., [35] proposes a new, more realistic model with a hierarchical platform view for global scheduling on unrelated processors and present several new workload assignment methods. We argue the global scheduling on multiple hosts is not feasible with the algorithms known today. A single ready task queue would exist in the cluster in such a case, and the ready state processors would select the tasks. Suppose the queue and processors are located on separate nodes. In that case, the added network delay between them will slow down the cluster's computing performance to a level that makes the system unusable. Hence, in this paper, we focus our efforts on partitioned scheduling.

Prior art on FaaS performance guarantees There exists research works like PyWren [36] ExCamera [37] and Sprocket [9] where the authors examine the low-latency serverless processing from the software developer's point of view. They assume that the serverless system such as Amazon AWS [38] is given, and the question is how to execute and scale the user demanded function to process the incoming data (e.g., video frames) as soon as possible by exploiting fine-grained parallel computations and elasticity. The proposed cloud-based frameworks control the fleet of workers, use the same generic Lambda function to avoid cold starts, and exploit the parallelization of functions to ensure real-time processing, e.g., in the case of video streams. We argue that these works can provide only soft real-time performance since they do not guarantee in the infrastructure level to process the data before the deadline. By parallelizing the user functions, the frameworks accelerate the data processing; however, if the used serverless FaaS

system is overloaded, either the parallelization is impossible [39], or a large number of functions do not finish fast enough [40], [41]. To sum up, the proposed frameworks use the FaaS service and thus the cloud infrastructure, but they cannot control it. In contrast to these low-latency solutions, in this paper, we examine the RT processing challenge from the FaaS provider's point of view. We argue about the necessity of an RT-FaaS system, which is able to control the underlying infrastructure, thus ensuring the hard real-time execution of user demanded functions.

Some works like [39], [40], [41], [42] propose modifications in FaaS platforms (e.g., OpenFaaS or OpenWhisk) to enable performance guarantees in FaaS systems. Authors of [39] examine how to guarantee the invocation rate of FaaS functions even in overloaded cases. However, they do not address the guarantees on the function executions. In [40], the authors concentrate on enabling execution SLAs, i.e., ensuring customer-provider function execution agreements. They introduce Function as a Service SLA Framework (FaaS2F), a modular framework to define execution-SLAs, and detect violations of them in serverless functions. It can detect the scenarios when a function execution is slower than in the case of normal FaaS load, but the users cannot define the required deadline for their functions. The authors of [41] propose a platform for running latency-sensitive serverless computations on edge resources. They provide an approach for allocating edge resources for latency-sensitive serverless functions and an algorithm to scale the resources dynamically. They aim for soft real-time executions and do not consider the latency of the network among the edge servers.

Our paper envisions a FaaS system where both RT execution and networking are ensured for RT functions, thus enabling hard RT performance. Furthermore, we propose a scheduling method for this system, where users can define the required deadline to finish their tasks and get back the computed output.

4 RT COMMUNICATION WITHIN THE RT-FAAS

We propose three conceptual designs to realize the RT communication of functions in the server cluster. The possible communication paths in the RT-FaaS are depicted in Fig. 1. As Def. 1 argues, the RT-FaaS operates on a cluster, consequently, the functions can run on multiple servers. In order to adapt to available FaaS systems and to meet Requirement 1 of RT-FaaS, we assume that functions run in containers. In Fig. 1, two applications exist (*RT app 1* and *non-RT app 2*) to which one-one function is deployed in the *RT Container 1* and the *Container 2* of the RT-FaaS. The application sends input data to the function, where the processing happens, and it sends the response back to the application. It might happen that two functions in different containers also communicate with each other. E.g., a third-party cloud service – for instance, a low-latency RT cloud database – could also run inside a container that the user-demanded functions call.

Within the nodes, a virtual network ensures the connectivity of the containers, like any Container Network Interface (CNI) plugin for Kubernetes pods. The nodes contain a single or multiple NICs, which connect the physical

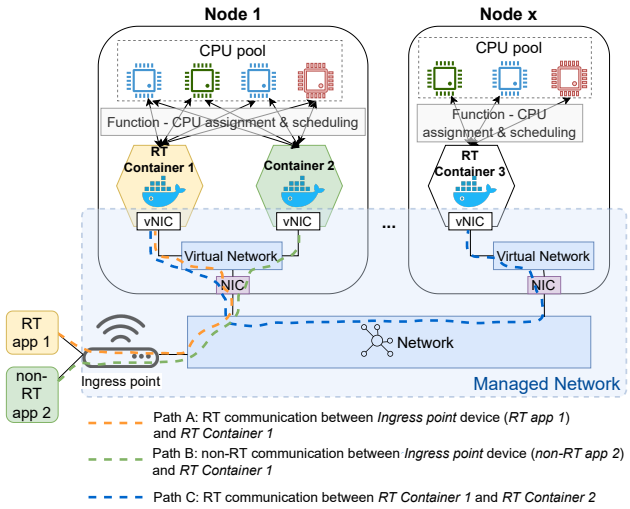


Figure 1: The architecture of the RT-FaaS

network devices to the virtual network of the servers. Another essential element is the ingress point through which the RT or non-RT applications can communicate with their deployed functions. This component is considered as the edge of the RT-FaaS managed network. Beyond this point, it is the user's responsibility to ensure the RT connection to its application.

The network between the RT application and its corresponding RT function needs to be deterministic to guarantee the RT communication. Since the network resources are shared, the bounded latency for RT traffic must be ensured even to the detriment of the concurrent non-RT traffic. The 802.1Q standard Tagging could be used to prioritize the RT traffic since the VLAN header contains priority bits. Consequently, tagging the Ethernet frames of the RT traffic results in RT packets being forwarded ahead of the others, but it cannot provide the E2E latency of the flow. If multiple devices use different system clocks, the packet buffering effect can result in unpredictable latency in the E2E traffic flow. TSN [14] tackles this issue. The three key components are i) Time synchronization of devices in the TSN domain, ii) Traffic shaping within the TSN devices, and iii) the selection of the communication paths and the reservation of them. If all TSN devices run synchronously and use the same traffic shapers, packet bursts will be prevented. However, TSN has initially been suitable for robot-controller communication on Ethernet networks, while in the RT-FaaS case, the TSN network endpoints are computing servers. Therefore, the servers' virtual network must be also deterministic, thus forming a common TSN domain with the physical network.

We propose (in Def. 4) the three conceptual design approaches how to make real-time, i.e., worst-case latency bounded, communications of the RT-functions within the nodes and connect them to the physical TSN network.

Definition 4. ***The conceptual design approaches*** *to implement the RT communications in RT-FaaS are:*

1) *Hardware-based real-time networking;*
2) *Software-based real-time networking;*
3) *Hybrid real-time networking.*

4.1 Hardware-based real-time networking

This design attaches TSN-enabled hardware devices to the servers. Accordingly, the server NICs form a common TSN domain with the hardware devices outside the servers, e.g., TSN switches. In this case, the TSN switches and NICs are synchronized by a "universal clock", and all of them use the same traffic shaper to schedule the traffic of the applications within the TSN domain to enable the required determinism.

Yi Wang, in presentation *Time-Sensitive Networking Enabling on StarlingX* [43], at the Open Infrastructure Summit 2020, elaborated on how they enable TSN in a StarlingX cluster for workloads running in Kata containers. StarlingX [44] is a complete cloud infrastructure software stack for the edge designed for ultra-low latency use cases. Kata containers [45] have their own kernel independently from the host's. For this reason, the user-demanded TSN functions must be in Kata containers to provide independence and isolation. They used four servers as a cluster, each containing TSN-supported Intel I210 series NICs, which connect to a TSN switch PCIe-0400-TSN. Tools *ptp4l* and *phc2sys* were used for time synchronization of the TSN devices. Data sender and data receiver TSN applications run in Kata containers on different nodes. The I210 NICs were linked into the Kata containers by PCI passthrough. They managed to achieve deterministic E2E latency (ranged from 1253.188μs to 1253.343μs), using a 2 ms packet sending period, while a large amount of concurrent non-RT traffic was sent over the TSN network, too. Their work is a great example of how to enable RT communication of RT-function containers in the RT-FaaS. However, the major drawback of this approach is that the NICs are not shared between the containers. Thus, the number of TSN-enabled NICs determines the upper bound for the number of deployable RT-functions within the RT-FaaS server.

Single Root I/O Virtualization (SR-IOV) could be an effective solution to share a single NIC to the containers within the same node. It allows a single PCIe hardware interface to be shown as multiple physical PCIe devices, i.e., containers can share a single physical NIC, which appears as separate physical devices. With the adequately configured BIOS and OS, SR-IOV supported NICs perform networking tasks from hardware, i.e., there is no need for an additional virtual bridge to connect NIC and containers. Utilizing SR-IOV could alleviate the upper bound limitation on the number of RT-functions; however, to the best of our knowledge, there is no available NIC device that supports both TSN and SR-IOV.

Another way is the utilization of smartNICs [46] within the RT-FaaS nodes. They offload the CPU by enabling network traffic processing on the NIC. SmartNICs allow outsourcing the virtual network within the nodes to the HW of the NIC.

4.2 Software-based real-time networking

Not only the physical devices need to operate in real-time, but the virtual networks within the RT-FaaS nodes as well. The Software-based RT design is about making virtual network software which is operated by the node's CPUs. No special purpose HW is required; the virtual network software connects the containers to the physical NIC and aims to guarantee the bounded latency of the packet transmissions. Research products already exist aiming at accelerating the performance of the virtualized networking. [47] introduces an software switch called ESwitch that can scale over 100 Gbps. The authors compared ESwitch to OVS (Open vSwitch) [48], the most commonly used software-based switch, and proved that ESwitch significantly outperformed OVS.

However, ESwitch still does not provide upper-bounded latency. Currently, it is an actively researched field to make virtual switches support TSN. The open research problem is to which extent a software switch and kernel network stack can keep the hard guarantees of deterministic bounds. The authors of [49] and [50] propose virtual switches that are capable of TSN with the IEEE 802.1Qbv traffic shaper.

4.3 Hybrid real-time networking

The third design that we propose to enable RT communications for RT functions is the Hybrid method. This design aims to i) simplify the virtual network component of the RT-FaaS nodes to a level that is already capable of guaranteeing the bounded latency of transmissions, and ii) delegate the remaining complex networking tasks to the TSN-enabled network devices. Recall that the architecture presented by Wang [43], detailed in Sec. 4.1, could be suitable for the RT-FaaS system. However, it has a significant constraint: the number of NICs of a node is the upper bound for the number of deployable RT functions. To solve this issue, the common NIC could be shared among the containers if it is ensured that the RT traffic belonging to the RT-functions is prioritized over the non-critical traffic of the non-RT functions.

[22] presents a solution where servers contain TSN-enabled NICs and use their HW Tx/Rx queues for different traffic priorities. Their approach exposes the HW queues as traffic classes, thus allowing for Linux `qdiscs` to be attached. The `qdiscs` are kernel packet buffers for the network packets which control when and how they are transmitted. The authors developed `cbs qdisc` for credit-based shaping (standard 802.1Qav) and `tbs qdisc` for time-based scheduling. They use regular sockets for transmitting data and steer the traffic from these sockets to the right HW transmission queues of the TSN supported NIC. They managed to achieve 468 ns average and 506 ns max latency using 1 ms packet sending period. They have found that the worst-case latency can be further improved if they avoid the dynamic memory allocation for packet buffers just like Björn Töpel and Magnus Karlsson presented in [23]. The AF_XDP new socket family allows zero-copy of the packets by the XDP.

The major issue of this approach is the RT functions run as processes within the RT-FaaS server and use sockets to steer their traffic into the proper priority HW queue of the NIC. This does not meet with the RT-FaaS Requirement 1 in Def. 3. To this end, the RT-function processes should run in containers which results in unpredictable and unnecessary delay due to the virtual NIC and the additional network stack inside the container. Fortunately, Nakamura et al. [51] propose a novel approach for fast container networking that enables container applications to utilize the host network

stack directly with proper access control, bypassing the container's network stack and virtual interface.

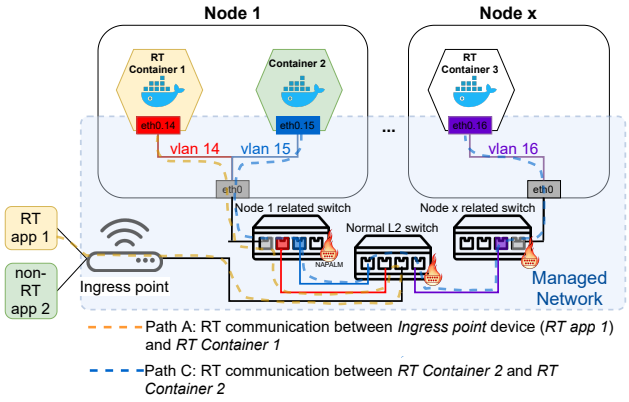


Figure 2: Hybrid RT networking design: *VLAN ID forwarding*

We further propose another realization of the Hybrid real-time networking which forwards packets inside the servers based on their VLAN ID, depicted in Fig. 2. In this architecture, the virtual network component is simplified to forwarding by VLAN ID only. Each container is located in a separate VLAN, so their outcoming packets are extended with the VLAN header. Each RT-FaaS node is paired with a physical switch, where the same VLANs are configured that the containers use inside the server. The VLAN trunk port connects the node-related switch and the node. This way, the virtual interfaces of the containers appear at the physical ports of the switch, which need to be connected – e.g., by an L2-switch – to enable the communication between the nodes and the ingress point. In this architecture, all the complex network tasks could run in the learning switch device. To support the dynamism of the containers' lifecycles, the VLAN configurations must be handled automatically. To this end, a CNI network plugin should be implemented to configure the VLAN within the nodes and in the switch devices by NAPALM [52]. The limitation of this design is on the number of deployable RT functions within the RT-FaaS server. As many RT functions could run in a node as many physical Ethernet ports exist in the node-related switch (due to the containers being routed out there).

5 Real-time Execution of the Functions

Unfortunately, RT communication on its own is not enough for response data to arrive back consistently in time, since it is not sure the function can utilize any of the node CPUs immediately when it is triggered. The RT function execution must be ensured too (implemented by the *Function-CPU assignment & scheduling* components of Fig. 2). The question is how to schedule RT and non-RT functions simultaneously such that deadlines are met, and non-RT tasks are not starved. We assume the RT-FaaS has the goal of scheduling the user demanded functions according to Def. 5.

Definition 5. *The objective of the proposed RT-FaaS scheduler is to assign RT functions to a minimum number of CPUs of the server cluster, such that the deadline requirements are met.*

Although a processor may run both RT and non-RT functions, as Brosky argues in [53]: *a shielded CPU makes it possible to guarantee rapid response to external interrupts and to provide a more deterministic environment for executing real-time tasks.* The objective function in Def. 5 leaves as many processors as possible to process the remaining non-RT functions, thus, ensures they will not starve for CPU cycle. In this paper, we consider a partitioned-EDF scheduling system for RT-FaaS and propose a partitioning algorithm that considers the heterogeneity of the processors and attempts to minimize the number of utilized CPUs for the RT functions. Furthermore, we assume that RT-FaaS operates on a deterministic network (the proposed concepts are presented in Sec. 4) both among the hosts and within them between the NIC and the RT function. Although deterministic networking is an actively researched topic, we believe that multiple RT network implementations will soon be available.

5.1 Deployment and Function model

We model the deployment that RT-FaaS users may request to deploy. The deployment process includes i) selecting the host and its CPU to deploy the container which runs the function, ii) configuring the network between the ingress point and container, and iii) scheduling the function in an RT manner.

Definition 6. *A deployment is modeled by three parameters (τ, R, T), where τ is the function – i.e., the source code to run – R is the response time(s) of the function, defined in Def. 7, and T is the trigger period of the function.*

Definition 7. ***Response time of a function:*** *The necessary time for i) the input data reaching the target host, ii) function being executed, iii) performing its cloud service calls, and iv) sending its response back to the ingress point.*

In this paper, we consider *implicit deadlines*, which means each function's deadline equals its period. Assuming that hosts in the infrastructure contain heterogeneous multiprocessors, R includes multiple response times, one for each processor. If n deployment requests arrive simultaneously, the RT-FaaS needs to schedule n functions on the managed cluster and configure the network routes forth and back to the target server where functions are running. Please note we use terms *function* and *task* interchangeably because, in FaaS terminology, the former is used while RT scheduling terminology prefers the latter. Each deployment contains a function τ for which we have used the function model defined in Def. 8. The RT functions are assumed to be *periodic tasks*, i.e., all of them are invoked at regular intervals and repeat themselves after a fixed time interval. Each RT function is defined by three parameters.

Definition 8. *A function is determined by $\tau = (C, T, D)$, where C is the worst-case execution time (WCET), T is the release period, and D is the deadline for the execution.*

The tasks' CPU utilization can be calculated by $u = \frac{C}{D}$. Similarly to the deployment model, if hosts in the infrastructure contain heterogeneous multiprocessors, the parameter C and D are also a set of possible elements depending on the processor on which the task is processed.

5.2 Relations between deployment and function model

Naturally, the users cannot define all parameters related both to the deployment and function models. E.g., let us assume a deployment request which includes an object detection algorithm as a function. It detects objects on the input video frames, writes the number of detected elements to a DB – through a real-time cloud DB service – and also returns this value to the user. Most likely, the user can define only i) the source code of the function, ii) the WCET c of this algorithm without any cloud service call, iii) the frame rate of the input video stream, and iv) the RT cloud service used by the uploaded function.

In this example, the frame rate of the input video is equivalent to the trigger period T of the deployment, while the RT cloud service is the DB writing. The response time R of the deployment depends on the location and the performance of the CPU which executes the deployment's task τ. Moreover, the example function takes advantage of the DB writing RT cloud service, so the worst-case execution time of the task becomes $C = \omega(t_{RT\_service}) + c$, where $t_{RT\_service}$ is the time of calling the applied RT cloud service, ω is the number of the calls, and c is the time needed to run the function code. So far, the worst-case execution time of the demanded function is known, but we also must count with the network effect of the RT-FaaS to determine its response time. The response time of the deployment shall be $R = t_{i\to f} + C + t_{f\to i}$, where $t_{i\to f}$ and $t_{f\to i}$ are the transmission times between the ingress point and the function. While c and ω are defined by the user, in a distributed system, both the transmission times and the real-time cloud service execution times may depend on the CPU where the deployment's functions are running. Consequently, these transmission and execution times rely on the partitioning of the requested functions.

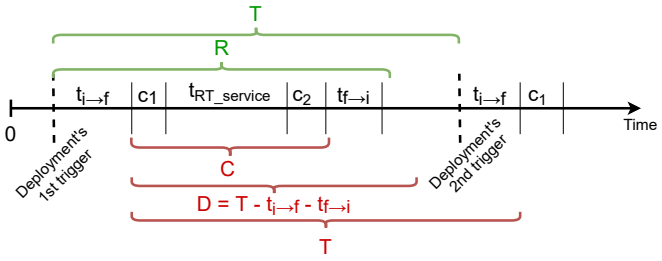


Figure 3: Deployment and function model relations.

For an illustration of these parameter relations, let us examine Fig. 3. The axis shows the timeline of the previously mentioned periodically triggered object detection function. We depict the function and the deployment-related parameters in red and green colors. This example assumes that $\omega = 1$. T tells us how frequently a video frame arrives at the RT-FaaS ingress point, which triggers the function to process it. R is the response time for the trigger, which is obtained by summing the network-related transmission times $(t_{i\to f}, t_{f\to i})$ and the worst-case execution time C of the task. We have divided C into c_1 (before the RT service call) and c_2 (after that). Needless to say, though in this example, we have assumed only one service call, the function could use multiple services and access them more than once. We assume that the task reserves the processor during the external real-time DB access, i.e., the $C = c_1 + c_2 + t_{RT\_service}$. The deadline for the task is the deadline for the deployment trigger's response time without the transmission times, i.e., $D = T - t_{i\to f} - t_{f\to i}$. This way, it is guaranteed the processed task's return value has enough time to get back to the ingress point. Since we assume deterministic underlying network, $t_{f\to i}, t_{RT\_service}$ are all constants, and the RT-FaaS platform is aware of their values.

5.3 Infrastructure model

We assume the RT-FaaS infrastructure is a data center topology containing multiple hosts, which altogether include m CPUs: $\pi_1, \pi_2, \ldots, \pi_m$. The ingress point location – through which the deployed functions are available from outside – is known beforehand. Assuming that i) the RT-FaaS platform is aware of network bandwidths and latencies within the cluster, and ii) knows the necessary execution time of real-time cloud service calls, we can form the *response time matrix*:

$$\begin{array}{c} \pi_1 \quad \pi_2 \quad \ldots \quad \pi_m \\ \begin{pmatrix} r_{1,1} & r_{1,2} & \ldots & r_{1,m} \\ r_{2,1} & r_{2,2} & \ldots & r_{2,m} \\ \ldots & \ldots & \ldots & \ldots \\ r_{n,1} & r_{n,2} & \ldots & r_{n,m} \end{pmatrix} \end{array}$$

The columns represent the CPUs of the managed cluster, while the rows correspond to the deployments that the users want to deploy. One cell means the response time of the deployment if its task is assigned to the regarding processor. E.g., if the second deployment's function τ_2 is executed by processor π_1, then its response time for a trigger is $R_2 = r_{2,1}$.

One might notice, we have not considered the multiple CPUs in the model parameter calculations, so far. More than one CPU results in the function parameters C and D are vectors, i.e., the RT-FaaS calculates i) the execution time of τ_1 task for all processors $C = \{C^{\pi_1}, C^{\pi_2} \ldots, C^{\pi_m}\}$ and ii) the task's deadline requirements $D = \{D^{\pi_1}, D^{\pi_2} \ldots, D^{\pi_m}\}$. Parameters $C^{\pi_j} = \omega(t^{\pi_j}_{RT\_service}) + c$ and $D^{\pi_j} = T - t_{i\to\pi_j} - t_{\pi_j\to i}$ for each $1 \leq j \leq m$. Thus, we obtained the model of the function $\tau_1 = (C, T, D)$. Then the RT-FaaS also can determine the elements of the response time matrix related to the τ_1: $r_{1,j} = t_{i\to\pi_j} + C^{\pi_j} + t_{\pi_j\to i}$ for each $1 \leq j \leq m$. Thereby the response time of the user's deployment request relies on the CPU set: $R = \{r_{1,1}, r_{1,2}, \ldots, r_{1,m}\}$. Thus, the model of the deployment is known as well: (τ_1, R, T). By now, one might notice it is great challenge of how to assign the requested tasks to the managed CPUs of the RT-FaaS in order to meet their deadlines. We formalize this in Sec. 5.4.

5.4 ILP formalization of task partitioning

The input of the task partitioning problem on RT-FaaS is the $I = (n, m, M)$ triplet, i.e., the number of tasks n and CPUs m, and the utilization matrix M. The utilization matrix is similar to the response time matrix: the columns represent the CPUs, while rows correspond to the user's tasks. On cell of the utilization matrix (e.g., $u_{1,2}$) represents the CPU utilization $C_1^{\pi_2}/D_1^{\pi_2}$ of task τ_1 if it is assigned to π_2. Our objective – see Def. 5 – is to find a feasible partitioning –

using the least number of processors – of the given task set such that their deadline requirements are met.

The ILP formalization of the problem contains two variable types, $x_{\pi_j}^{\tau_i}$ and y_{π_j} for $1 \leq i \leq n$ and $1 \leq j \leq m$. $x_{\pi_j}^{\tau_i}$ denotes whether τ_i is assigned to π_j, while y_{π_j} indicates if any of the tasks were assigned to π_j.

$$\text{minimize} \sum_{j=0...m} y_{\pi_j} \tag{1}$$

subject to

$$\forall \tau_i \in \tau : \sum_{j=1...m} x_{\pi_j}^{\tau_i} = 1 \tag{2}$$

$$\forall \pi_j, j = 1...m : \sum_{i=1...n} \frac{C_i^{\pi_j}}{D_i^{\pi_j}} \times x_{\pi_j}^{\tau_i} \leq 1 \times y_{\pi_j} \tag{3}$$

$$\forall\, j = 1 \ldots m, i = 1 \ldots n : x_{\pi_j}^{\tau_i}, y_{\pi_j} = \{0, 1\} \tag{4}$$

Variables are $x_{\pi_j}^{\tau_i}$ and y_{π_j} where $i = 1, ..., n$ and $j = 1, ..., m$. $x_{\pi_j}^{\tau_i} = 1$ if τ_i is assigned to π_j otherwise it is zero. Furthermore, $y_{\pi_j} = 1$ if π_j is used, otherwise it is zero.

The constraints are the following. Each task must be assigned to exactly one host (2). The cumulative utilization of tasks assigned to the same CPU must be less or equal than one due to EDF's schedulability requirement (3). Partitioned scheduling on heterogeneous multiprocessors is intractable in polynomial time, and it is an $\mathcal{NP}$-hard problem, even if processors are identical [54]. In fact, Lenstra et al. [55] have proved that no algorithm is able to find a solution in polynomial time with a smaller approximation ratio than $3/2$.

5.5 Scheduling model

Once we know the deployment, function, and infrastructure models, the only remaining question is how to schedule the requested functions, i.e., define a scheduling model that determines where and when to execute the tasks. We argue that one possible solution could be the *partitioned EDF model*, which guarantees that the tasks assigned to a particular CPU will finish before their deadline if the sum utilization is less than one. E.g., if three tasks τ_1, τ_2 and τ_3 are assigned to π_1 then it is guaranteed to meet their deadlines if $\sum_{i=\tau_1...\tau_3} \frac{C_i^{\pi_1}}{D_i^{\pi_1}} \leq 1$. This is taken care of by constraint (3) of the ILP problem. Consequently, the scheduling problem is the following. The input includes the set of deployment requests, i.e., the deployment and function models and the infrastructure model. Applying the partitioned EDF scheduling method, we first need to partition the tasks among the CPUs, which is the spatial optimization part of the scheduling problem. Second, for each CPU, the well-known EDF algorithm fulfills the tasks' temporal scheduling, i.e., it decides when to start them.

The question may arise as to whether using a real-time scheduler instead of a CFS scheduler affects performance. Even though, our proposed partitioned-EDF scheduling scheme uses RT schedulers (EDF) for each CPU that executes RT functions, we believe that for running the OS, the FaaS platform, and the non-RT functions, the default Linux CFS scheduler could be responsible. Within the FaaS servers, some CPU resources must be allocated to the CFS to avoid server failures. To strike a balance between RT FaaS applications and other non-RT applications, we adjust our proposed partitioning algorithm's objective function to use the minimum number of CPUs to run RT tasks. This maximizes the utilization of RT CPUs and prevents non-RT functions from starving for available CPUs or workers.

6 Task Partitioning Algorithm

We propose an approximation algorithm – denoted by ALG – that distributes the given task set among the CPUs of cluster nodes. The input is the same as for the ILP, i.e., the $I = (n, m, M)$ triplet. Its steps are the following:

1) Convert M to a boolean matrix M' according to the transformation rule defined in Def. 9.
2) Iterate through the columns of M' and calculate their weights $w(\pi_i)$, i.e., the number of ones in the column.
3) Consider the best processors in M' (see Def. 10).
4) Sort the columns of M' according to their weights in decreasing order.
5) If *exclusive task* exists (defined in Def. 11):
   a) Pick its processor in M', regardless its weight, and assign all the one elements (tasks) of the processor's column to the processor
   b) Delete the assigned tasks and processor, i.e., remove the proper rows and column from M and start again with Step 1 if still row exists.
6) If an unassigned task exists:
   a) Pick the processor in M' with the maximal weight – if multiple exist choose randomly – and assign the one elements (tasks) to it in the partitioning.
   b) Delete the assigned tasks' rows and the processor's column from M and start again with Step 1.
   c) Return the task-processor assignments

The return value of ALG is an assignment of functions to processors such that the EDF schedulers of the processors can guarantee the real-time completion of them.

Definition 9. ***Converting M to boolean matrix M': 1)*** *Sort the elements with their original indices in increasing order in each column π_i.* ***2)*** *Iterate through the cells $j = 1, ..., n$ in each column π_i, and set the element to 1 if $\sum_{k=1}^{j} s_{k,i} \leq 1$, otherwise set it to 0.* ***3)*** *Finally reorganize the cells according to their original indices. For an example see Fig. 4.*

Definition 10. ***Picking the best processors:*** *Given two processors π_1 and π_2 of the examined boolean matrix M'. It is said that π_2 is a better processor than π_1 if and only if all 1 items in π_1 are also 1 in π_2 and $w(\pi_2) > w(\pi_1)$. As π_2 is a better processor than π_1, the latter is removed from the columns of M'. We make this comparison for all column pairs.*

Definition 11. ***Exclusive task:*** *Such a task that could be run by only one specific processor.*

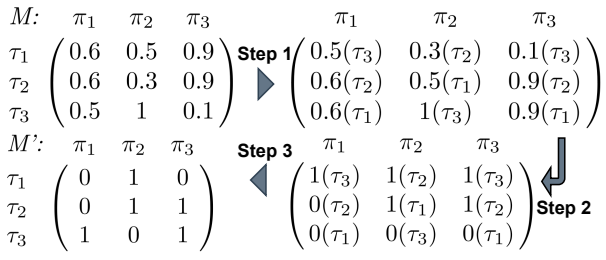


Figure 4: Example for generating M' boolean matrix

The proposed partitioning algorithm ALG is not tailored uniquely to FaaS. It can be also used in more general distributed systems (e.g., in a distributed video processing framework) without assuming the underlying FaaS platform. However, this paper aims to present how to enable RT execution in cloud computation systems. As we argue in Sec. 2, for that purpose, FaaS is the best choice since the cloud providers have the greatest possible control over their infrastructure with FaaS. Furthermore, the high-level FaaS user interface prevents the physical cloud infrastructure from being misconfigured by the customers. That is why we targeted FaaS systems and designed our proposed partitioning algorithm to be compatible with such a system.

6.1 Time complexity analysis

We produce the time complexity analysis of the proposed ALG to get a picture about the number of steps required in the worst-case scenario. Step 1 sorts the elements of each column of M, that takes $\mathcal{O}(mn\log(n))$ as any comparison sorting algorithm cannot perform better than $\mathcal{O}(n\log(n))$ [56]. Then, iterating through m columns and deciding for each element whether it is 1 or 0 takes $\mathcal{O}(mn)$ time. All in all, step 1 has $\mathcal{O}\big(mn\log(n)\big)$ complexity. Step 2 and 3 requires $\mathcal{O}(mn)$ iterations. Sorting the columns according to their weights takes $\mathcal{O}\big(m\log(m)\big)$. Finding the exclusive task in step 5 and iterating through its processor's column both require $\mathcal{O}(n)$. Choosing the maximum weighted column and checking its tasks to be 1 or 0 takes again $\mathcal{O}(n)$ in step 6. Finally, the six steps are need to be repeated in the worst case $\mathcal{O}(m)$ times.

Theorem 1. *The time complexity of ALG is* $\mathcal{O}\Big(m^2\big(n\log(n)+\log(m)\big)\Big)$.

6.2 Optimal corner cases

We examined the scenarios when the proposed algorithm returned optimal partitioning. In the following, $OPT(I)$ indicates the number of used CPUs in the optimal partitioning for input I given by the ILP solver of the problem defined in Sec. 5.4. Similarly, let $ALG(I)$ depict the number of used processors given by the proposed heuristic algorithm *ALG*.

Lemma 1. $OPT(I)=1 \iff ALG(I)=1$

Proof. If there exists a CPU π_i that is able to execute all tasks, guaranteeing their deadline requirements, then the corresponding column in M' contains only ones, i.e., $w(\pi_i)=n$. Since the weight of a column is $0 \le w(\pi_j) \le n$, no column exists with higher weight than n. *ALG* in the step 4 sorts this optimal column to the first place, i.e., in the next step this CPU will be visited. Thus *ALG* also finds the optimal π_i. □

Lemma 2. *In the worst-case scenario ALG visits all CPUs (from 0 to m) during the execution.*

Theorem 2. *If M size is $n \times 2$ then $OPT(I) = ALG(I)$.*

Proof. If $OPT(I)=1$, then $ALG(I)=1$ (Lemma 1). If $OPT(I)=2$, then due to Lemma 2, $ALG(I)=m$ and in this case $m=2$. □

Assumption 1. *Suppose the CPUs used in the optimal partitioning can be read from matrix M': using the Boolean algebra's OR binary operation between the optimal CPU columns in M' results in an all-ones column vector.*

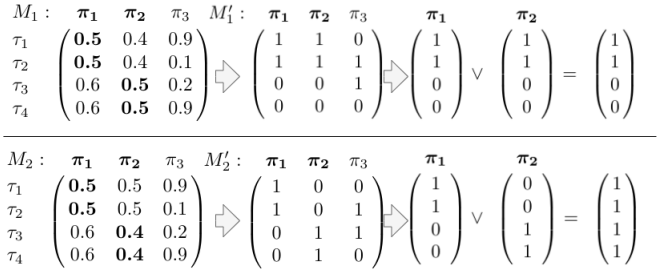


Figure 5: Two input cases: when optimal CPUs cannot (M'_1) and can (M'_2) be read from M'

In Fig. 5, two different utilization matrices M_1, M_2 are illustrated. The optimal assignments are the same for the two cases: $\pi_1 \leftarrow \{\tau_1,\tau_2\}$ and $\pi_2 \leftarrow \{\tau_3,\tau_4\}$. In case of M_2, Assumption 1 is true, since all cells of the optimal assignments contain ones. In contrast, from the boolean version of M_1, we could not tell that π_1 and π_2 are used in the optimal assignment, since the cells (τ_3,π_2) and (τ_4,π_2) are zeros.

Theorem 3. *If Assumption 1 is true for the input M, the problem of task partitioning (Sec. 5.4) is still $\mathcal{NP}$-hard.*

Proof. The task partitioning problem with Assumption 1 relaxation is equivalent with the Set Cover Problem: given a universe $\mathcal{U}=\{\tau_1,\tau_2,\ldots,\tau_n\}$ including all tasks, and a family $\mathcal{S}$ of subsets of $\mathcal{U}$. A subset is a column of M' containing the tasks with one elements, e.g., using M'_2 of Fig. 5 the subsets are $\mathcal{S}=\{\{\tau_1,\tau_2\}_{\pi_1},\{\tau_3,\tau_4\}_{\pi_2},\{\tau_2,\tau_3\}_{\pi_3}\}$. A cover is a subfamily $\mathcal{C}\subseteq\mathcal{S}$ of sets whose union is $\mathcal{U}$. The minimum set cover optimization problem is looking for the minimum number of sets whose union is $\mathcal{U}$, i.e., it returns the minimum number of CPUs that guarantee the real-time execution of all tasks. Since the optimization version of set cover problem is $\mathcal{NP}$-hard [57] the task partitioning problem with Assumption 1 relaxation is $\mathcal{NP}$-hard too. □

Theorem 4. *If $OPT(I)=m$ and Assumption 1 is true $\Rightarrow$ $OPT(I)=ALG(I)=m$.*

Proof. $OPT(I)=m$, i.e., there is no partitioning that uses less than m CPUs and due to Lemma 2, $ALG(I)=m$ in the worst-case $\Rightarrow OPT(I)=m=ALG(I)$. □

Lemma 3. *$OPT(I) = 2$, Assumption 1 is true and ALG starts visiting with an optimal CPU $\Rightarrow$ ALG in the next iteration will find the other optimal CPU, i.e., $OPT(I) = ALG(I) = 2$.*

Lemma 4. *If $I = (n, m, M_{n\times m})$ task partitioning problem is given, $OPT(I) = x$ where $(1 < x < m)$, Assumption 1 is true and ALG starts visiting with an optimal CPU $\Rightarrow$ the original problem is reduced to an $I' = (n_1, m-1, M_{n_1\times m-1})$ partitioning problem where $n_1 < n$ and $OPT = x - 1$.*

Lemma 5. *If Assumption 1 is true and there is no exclusive task in $M' \Rightarrow OPT(I) \leq \lceil \frac{m}{2} \rceil$.*

Proof. There is no exclusive task means that at least two ones are in each row of M'. Consequently, the number of ones in $M' \geq 2n$. Either way ALG picks a CPU column of M' to visit, due to Assumption 1, the remaining tasks can be assigned to the remaining CPUs of the next iterated M' (obtained by Step 5b or Step 6b of ALG). Since the initial boolean matrix M' does not contain exclusive task, the reduced M' matrices does not include them either. In the worst case, only $2n$ ones are included in M' and ALG reduces the matrix $\frac{m}{2}$ times. □

Lemma 6. *If Assumption 1 is true and there is no exclusive task in M' and $OPT(I) = \lceil \frac{m}{2} \rceil \Rightarrow ALG(I) = OPT(I)$.*

Proof. According to Lemma 5, if in the worst-case $OPT = \lceil \frac{m}{2} \rceil$. If in each iteration, the optimal CPU column has the the highest weight, ALG also choose them resulting in $ALG(I) = OPT(I)$. If would exist a column with a higher weight then the max optimal's, then in the next iteration M' would contain less rows resulting in finding a partitioning solution less then $\frac{m}{2}$ steps, which contradicts Lemma 5. □

Theorem 5. *If the size of M is $n \times 3$ and Assumption 1 is true, then $OPT(I) = ALG(I)$.*

Proof. If $OPT(I) = 1$, then $ALG(I) = 1$ (Lemma 1). If $OPT(I) = 3$, then $ALG(I) = m = 3$ (Theorem 4). If $OPT(I) = 2$, two scenarios could happen: exclusive item(s) exist(s) and do(es) not. In both cases $ALG(I) = OPT(I)$, due to Lemma 3 and Lemma 6.

□

Theorem 6. *If the size of M is $n \times 4$ and Assumption 1 is true, then $OPT(I) = ALG(I)$.*

Proof. In case of $OPT(I) = 1$ and $OPT(I) = 4$, due to Lemma 1 and Theorem 4, $ALG(I) = OPT(I)$. If $OPT(I) = 2$ and there exist some exclusive tasks, ALG will definitely start visiting with an optimal CPU, consequently, due to Lemma 3, $ALG(I) = OPT(I)$. If $OPT(I) = 2$ and exclusive tasks do not exist, due to Lemma 6, $ALG(I) = OPT(I)$. If $OPT(I) = 3$ and exclusive tasks exist, ALG will definitely start visiting with an optimal CPU, consequently, due to Lemma 4, the reduced problem in the next iteration will be $I' = (n_1, 3, M_{n_1\times 3})$ and $OPT = 2$ in which $ALG(I) = OPT(I)$, due to Theorem 5. If $OPT(I) = 3$ and exclusive tasks would not exist, such a scenario could not happen, since it would contradict Lemma 5.

□

We believe that in small edge FaaS infrastructures (e.g., physical computation servers in a mobile base station), a four-core system can be a realistic serverless environment. In these cases, Assumption 1 is true, the proposed ALG always performs optimal partitioning.

6.3 Approximation rate of ALG

Algorithm 1 Get Worst-Case result of ALG

```
1:  procedure WC_ALG(w_opt = {w_1, ..., w_i})
2:      used_cpu_count = 0
3:      next_iter_weights = w_opt
4:      while next_iter_weights ≠ {0, 0, ..., 0} do
5:          next_iter_weights = WC_STEP(w_opt)
6:          used_cpu_count += 1
7:      return used_cpu_count
8:  procedure WC_STEP(w_opt)
9:      max_value = max{w_opt}
10:     for i in{1, ..., max_value} do
11:         index_of_max = get_index_max(w_opt)
12:         w_opt[index_of_max] -= 1
13:     return w_opt
```

Given n, m and $OPT(I)$, our goal is to determine how much ALG deviates from the optimum in the worst case. Suppose the optimal partitioning is *a priori* known, i.e., the weights of the optimal CPU columns in M' are available in vector $\vec{w}_{opt}$. ALG, in the worst-case, visits these columns for the last time, i.e., in each iteration, it visits a CPU π_i column which has weight $w(\pi_i) \geq max\{\vec{w}_{opt}\}$. In the worst case, this π_i column's weight $w(\pi_i) = max\{\vec{w}_{opt}\}$ and contains ones from the optimal CPU columns so that in the next iteration i) most of the optimal CPU columns remain in M' ii) and they contain as many one elements as possible. Let WC_ALG mean the number of used CPUs by ALG in the worst-case, calculated by Algorithm 1.

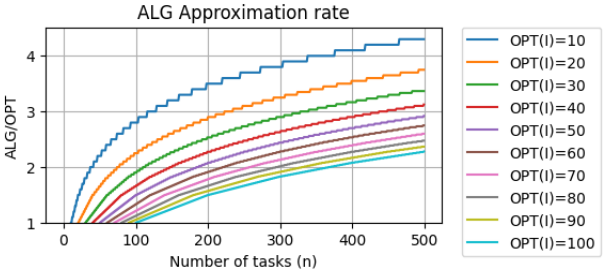
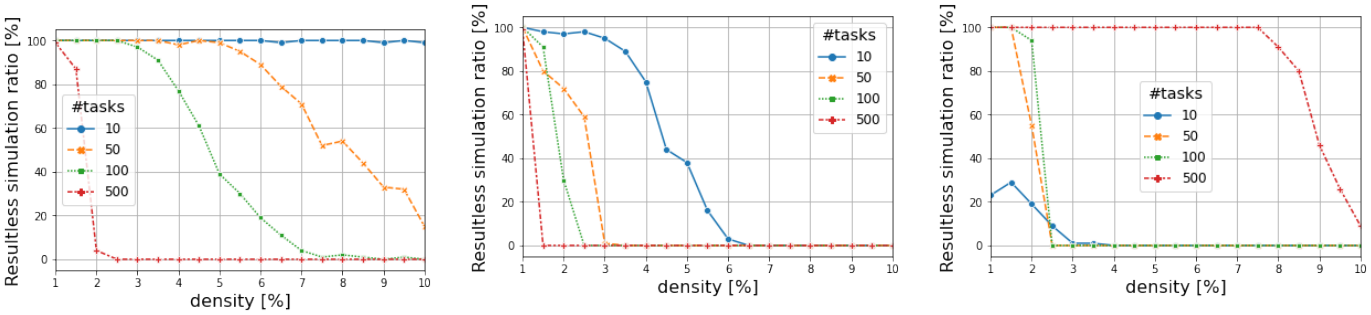


Figure 6: Approximation rate of ALG

We have calculated the largest differences between ALG and OPT using Algorithm 1 in case of $m = 100$ and $n = 10, \ldots, 500$, which are depicted in Fig. 6.

7 Evaluation

To the best of our knowledge, there is no publicly available RT-FaaS system that would enable hard real-time executions. Therefore, it is not possible to directly compare our proposed RT-FaaS design to another approach. Instead, we focus on the evaluation of the proposed partitioning algorithm. As we have elaborated in Sec. 5, we aim to utilize



(a) Ratio of *BR* simulations without result

(b) Ratio of *EBR* simulations without result

(c) Ratio of *FFD* simulations without result

Figure 7: Percentage of the simulations when the examined (a) *BR*, (b) *EBR* and (c) *FFD* algorithms have no solution. Our proposed algorithm *ALG* always returns a feasible partitioning result.

the minimum number of CPUs for RT tasks to provide the remaining ones for non-RT functions. Due to the dynamic nature of the cloud, we argue that it is essential for RT scheduler i) to consider a wide variety of heterogeneous processors in a multi-node cloud system, ii) to run fast (in polynomial time) to minimize scheduling overhead, and iii) not to migrate tasks among nodes. As far as we know, the most recent real-time scheduler studies assume a single node environment, meaning that they either consider identical CPUs/cores [26], [27] or a limited degree of heterogeneity for the processing entities [29], [30]. These schedulers are not comparable with our partitioned-EDF design since the various task runtimes on multiple different cores/processors cannot be modeled. That is why, we use Barauh's algorithm (*BR*) [34] and the well-known First Fit Decreasing (*FFD*) [58] bin-packing offline heuristic algorithm as a comparison basis to our proposed approach. *BR* is based on LP-relaxation of the original ILP problem and aims to minimize the maximum fraction of the capacity of any CPU that is utilized to process RT tasks. One might notice, this objective is quite the opposite of what we aim for: *BR* returns a feasible partitioning that uses as many CPUs as possible to minimize the maximum cumulative utilization. To direct *BR* to find the minimum number of CPUs, we wrapped it into a for cycle. Hereinafter we refer to this extended version as *EBR*. In each iteration ($i = 0, \ldots, m$), *EBR* selects the first i CPUs with the greatest weights and calls the *BR* that considers only them as possible targets for the tasks. The weights are calculated as they were in step 2 of *ALG*. If *BR* algorithm yields a feasible solution, then *EBR* breaks the cycle and returns.

7.1 Simulation inputs

The simulator we have used is available at https://github.com/hsnlab/partitioned_scheduling_on_RT-FaaS. The input for a simulation is the previously known $I = (n, m, M)$ triplet. We have used $n = 10, 50, 100, 500$ tasks and $m = 100$ CPUs. To generate a utilization matrix for the simulations, we have used the transformation, specified in Def. 9, inversely that works as follows. Let us assume a given $a \geq n$ as the number of the ones of the boolean matrix. We start with an $n \times m$ zero matrix and place a one into a randomly chosen cell of each row. Then we place the remaining $a - n$ ones to the matrix randomly. Next, we iterafte through the columns. Let us assume the column contains b ones, we set each cell to $\frac{1}{b}$ where originally a one was placed. Then, we replace the remaining zero values with a random number between $\frac{1}{b} + 0.05$ and 1. Such an obtained utilization matrix guarantees each task can be run by at least one CPU. Hereinafter instead of a we use the *density* feature of the boolean matrix (Def. 12).

Definition 12. ***Density feature of the boolean matrix:*** *a density function that returns the percentage of non-zero cells of the boolean matrix. If the density is* 0% *then it is a null matrix, while* 100% *density means an all-ones matrix.*

0% density means there is no CPU on which any of the input tasks could run in RT. The 100% density denotes all tasks could be scheduled on all CPUs. Our simulations answer questions such as *i)* how many times the algorithms do not return, *ii)* if solutions exist, what percentage of the available CPUs are used, *iii)* what is the runtime complexity of the algorithms, and finally, *iv)* what are the answers to *i* and *ii* assuming identical multiprocessors.

7.2 Number of feasible solutions

In Fig. 7 we depict the algorithms *BR*, *EBR* and *FFD* to show how many times they have not returned any feasible partitioning solutions. *ALG* is not displayed since it always returned a partitioning in our simulations. The *x-axes* show the density of the boolean matrix from which the utilization matrix was generated for the simulation. We have executed one hundred simulations for a given density, each of them with different utilization matrices. The *y-axes* depict the percentage of simulations when the algorithms have no feasible results. In the case of *BR* we conclude that the more tasks are waiting for partitioning, the better the possibility to find a valid partitioning. Similarly, this finding is true for *EBR*, furthermore, in its case, we are getting more results at a lower feasibility value. In contrast, the *FFD* bin-packing heuristic algorithm provided results for a large number of tasks starting only from 7.5% density value.

7.3 Number of provisioned processors for RT tasks

In Fig. 8 we present how the algorithms perform in task partitioning to minimize the number of CPUs for RT functions.

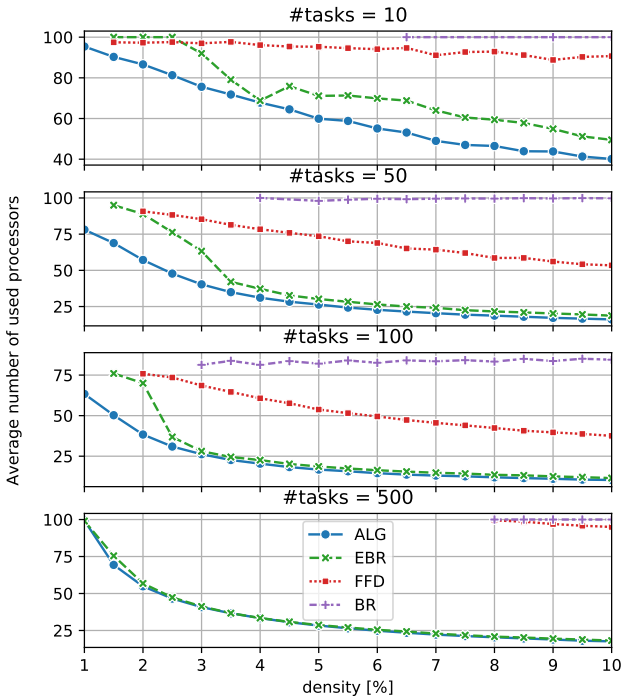


Figure 8: Average number of used processors for RT tasks in the partitioning depending on density and the number of tasks.

For a given boolean matrix density, we have executed one hundred simulations with different utilization matrices. Consequently, one point in the figure represents the average number of the used CPUs of the partitioning depending on the density value. In general, the *BR* algorithm performs the weakest, regardless of the task count. This is expected, as *BR*'s objective is the exact opposite of ours: it attempts to spread RT functions over the cluster using as many CPUs as possible. On the other hand, the *EBR* – that we have created to transform the original objective to find partitioning on the minimum number of CPUs – performs significantly better. In the best-case *ALG* outperforms *EBR* by 55 % ($\#tasks = 50$, density 2%), however as the task count increases, *EBR* provides almost the same performance as our proposed *ALG*. Finally, the bin-packing algorithm *FFD* performs between the efficient solutions (*ALG*,*EBR*) and the inefficient one (*BR*). Inferred from our simulations, the performance of *FFD* linearly decreases as the density increases. However, its success largely depends on the number of input tasks. E.g., with ten tasks, a solution is provided even for 1% density value. On the other hand, for 500 tasks, a valid solution is obtained only for $> 7\%$ density.

7.4 Runtime complexity

Here we examine the worst-case time complexity. As seen in Theorem 1, the time complexity of *ALG* is $\mathcal{O}(m^2(n\log(n) + \log(m))) \approx \widetilde{\mathcal{O}}(m^2 n)$ (we use $\widetilde{\mathcal{O}}$ from [59] to hide polylog(n) and polylog$(1/\delta)$ terms from $\mathcal{O}$ form). The algorithm *BR* [34] includes three main steps: 1) solving the LP-relaxation of the heterogeneous multiprocessor partitioning problem, 2) determining which tasks have exactly one CPU mapping in the LP solution, and 3) for the remaining tasks constructing a bipartite graph to find exact mappings for them. Step 2 and 3 can be done in $\mathcal{O}(nm)$ and $\mathcal{O}(n+m)$ time. Finding solution for a Linear Programming can be done by Interior point algorithm for which the best theoretical result is $\mathcal{O}(\mathrm{N}^\omega \log^2(\mathrm{N})\log(\mathrm{N}/\delta)) \approx \widetilde{\mathcal{O}}(\mathrm{N}^\omega)$ [60], where δ is the relative accuracy, ω is the matrix exponent, and N is the number of variables in the LP problem. In our case $\mathrm{N} = nm$. This result was derandomized by Brand [59] thus providing a deterministic algorithm matches one of the fastest randomized bounds of $\widetilde{\mathcal{O}}(\mathrm{N}^\omega)$. This stated bound holds for the current bound on ω with $\omega \approx 2.38$ [61]. So the time complexity of *BR* is $\widetilde{\mathcal{O}}((nm)^{2.38}) + \mathcal{O}(nm) + \mathcal{O}(n+m) \approx \widetilde{\mathcal{O}}((nm)^{2.38})$.

	Time complexity
ALG	$\widetilde{\mathcal{O}}(nm^2)$
BR	$\widetilde{\mathcal{O}}(n^{2.38}m^{2.38})$
EBR	$\widetilde{\mathcal{O}}(n^{2.38}m^{3.38})$
FFD	$\widetilde{\mathcal{O}}(n)$

Table 1: Time complexities of the partitioning algorithms

For what concerns *EBR*, it calculates the weights of each column in the matrix – $\mathcal{O}(nm)$ –, sort them – $\mathcal{O}(m\log(m))$ –, and choose the first i columns – $\mathcal{O}(m)$ – to get the matrix M_i. Then the BR algorithm is run $\widetilde{\mathcal{O}}((nm)^{2.38})$. In the worst case scenario, these are repeated m times, accordingly, the overall time complexity of *EBR* is $\mathcal{O}(m)(\mathcal{O}(nm) + \mathcal{O}(m\log(m)) + \mathcal{O}(m) + \widetilde{\mathcal{O}}(n^{2.38}m^{2.38})) \approx \widetilde{\mathcal{O}}(n^{2.38}m^{3.38})$. The worst-case time complexities are summarized in Table 1.

7.5 The homogeneous multiprocessor scenario

We have investigated the scenario in which the partitioning algorithms work on a set of identical CPUs. Here, the response time of deployment is independent of the CPU to which it is mapped, consequently, in such a homogeneous utilization matrix, the row values are the same. We have used $n = 10, 50, 100$ tasks, $m = 100$ CPUs and generated the homogeneous utilization matrix as follows. We randomly select a utilization value from a uniform distribution between 0.1 and 1 for all the elements of the first row of the matrix. Then, we summarize the selected values so far in a variable sum and select the next random value of a uniform distribution between 0 and $min(m - sum, 1)$ for the next row. Finally, we repeated the second step for all n tasks.

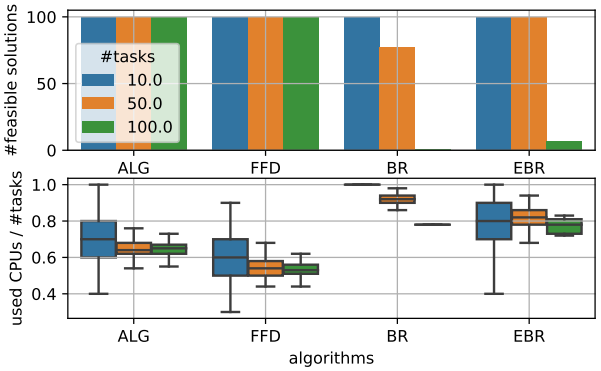


Figure 9: Algorithms in homogeneous multiprocessor scenario

The outcome is illustrated in Fig. 9. The upper part, shows how many times the algorithms return a feasible partitioning of the functions. For 10 tasks, all algorithms are able to find a solution in each simulation. Nevertheless, for 50 and 100 tasks, *BR* and *EBR* are not able to reach 100%. The bottom part of the figure shows the *goodness* of the algorithms: the number of used CPUs in the output partitioning normalized by the task count. Inferring from the results, the task count has no significant effect on the algorithms' performance. Furthermore, as expected, since this problem is equivalent to the Bin Packing Problem, the traditional offline heuristic algorithm *FFD* performs the best, followed by *ALG*, *EBR* and *BR*.

7.6 Evaluation Summary

The simulations suggest the proposed task partitioning algorithm has several advantages over the competition. ALG is the only algorithm that always found a feasible partitioning in our evaluations. It resulted in the lowest number of processors for partitioning. Although EBR was also effective for a large number of tasks, in terms of runtime complexity, it lagged behind significantly. We also studied the case of homogeneous utilization matrices where partitioning is equivalent to the classical bin-packing problem. Since FFD is designed for this problem, it gave the best outcome, but ALG came in second.

8 Conclusion

There has been a growing desire to expose time-critical and computational exhaustive applications to the cloud in recent years. Use-cases like remotely controlled robots or game streaming services require extremely low latency and real-time operation from the cloud platform to guarantee failure-free operation and high QoS. Our contributions are three-fold. First, we described a future RT-FaaS system that could be the next step of today's cloud approaches. Second, we proposed three design approaches to provide real-time communications in such a system and summarized the related work that could be the possible implementations of them. Finally, we proposed a real-time scheduling method, the partitioned-EDF with our partitioning algorithm, which can execute the RT functions. Assuming heterogeneous, multi-node multiprocessor scheduling, we presented that our algorithm outperforms the related prior approaches in the number of feasible solutions, number of provisioned CPUs, and the runtime complexity.

Acknowledgments

This work was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through projects i) no. 135074 under the FK_20 funding scheme, and ii) 2019-2.1.13-TÉT_IN-2020-00021 under the 2019-2.1.13-TÉT-IN funding scheme. L. Toka was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Supported by the ÚNKP-21-5 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

References

[1] S. K. Rao and R. Prasad, "Impact of 5g technologies on industry 4.0," *Wireless personal communications*, vol. 100, no. 1, pp. 145–159, 2018.

[2] J. G. Andrews *et al.*, "What will 5G be?" *IEEE j. sel. areas commun.*, vol. 32, no. 6, pp. 1065–1082, 2014.

[3] P. Varga *et al.*, "5g support for industrial iot applications - challenges, solutions, and research gaps," *Sensors*, vol. 20, no. 3, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/3/828

[4] M. Karrenbauer *et al.*, "Future industrial networking: from use cases to wireless technologies to a flexible system architecture," *at-Automatisierungstechnik*, vol. 67, no. 7, pp. 526–544, 2019.

[5] V. Struhár *et al.*, "Real-Time containers: A survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, p. 9.

[6] M. Cinque *et al.*, "Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, vol. 133. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019, pp. 5:1–5:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10742

[7] L. Abeni *et al.*, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[8] M. Szalay, P. Mátray, and L. Toka, "Real-time task scheduling in a faas cloud," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021.

[9] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 445–451.

[10] Daniel Bristot de Oliveira, "Deadline scheduling Part 1 - overview and theory," https://lwn.net/Articles/743740/, [Online; Accessed: 2021-2-16].

[11] K. Ramamritham, S. H. Son, and L. C. DiPippo, "Real-time databases and data services," *Real-time systems*, vol. 28, no. 2, pp. 179–215, 2004.

[12] A. Poniszewska-Maranda, R. Matusiak, N. Kryvinska, and A.-U.-H. Yasar, "A real-time service system in the cloud," *Journal of Ambient Intelligence and Humanized Computing*, vol. 11, no. 3, pp. 961–977, 2020.

[13] M.-R. Giovanny, T.-T. Alonso, M. Castillo-Cara, C. Blanca, and C. Carrión, "An experimental study of fog and cloud computing in cep-based real-time iot applications," *Journal of Cloud Computing*, vol. 10, no. 1, 2021.

[14] N. Finn, "Introduction to time-sensitive networking," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 22–28, 2018.

[15] N. Finn and P. Thubert, "Deterministic networking problem statement," *draft-finn-detnet-problem-statement-05 (work in progress)*, 2016.

[16] X. Yang, D. Scholz, and M. Helm, "Deterministic networking (detnet) vs time sensitive networking (tsn)," *Network*, vol. 79, 2019.

[17] A. Nasrallah *et al.*, "Ultra-low latency (ull) networks: The ieee tsn and ietf detnet standards and related 5g ull research," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 88–145, 2018.

[18] C. S. V. Gutiérrez *et al.*, "Real-time linux communications: an evaluation of the linux communication stack for real-time robotic applications," *arXiv preprint arXiv:1808.10821*, 2018.

[19] J. Kiszka and B. Wagner, "Rtnet - a flexible hard real-time networking framework," in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 1. IEEE, 2005, pp. 8–pp.

[20] Høiland-Jørgensen *et al.*, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.

[21] Gallenmüller *et al.*, "Comparison of frameworks for high-performance packet io," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2015, pp. 29–38.

[22] J. Sanchez-Palenci, "The Road Towards a Linux TSN Infrastructure," https://events19.linuxfoundation.org/events/elc-openiot-north-america-2018/program/schedule/, 2018, [Online; Accessed: 2021-6-13].

[23] M. Karlsson and B. Töpel, "Low-Latency, Deterministic Networking with Standard Linux using XDP Sockets," https://events19.linuxfoundation.org/events/embedded-linux-conference-europe-2019/program/schedule/, 2019, [Online; Accessed: 2021-6-13].

[24] C. Li *et al.*, "Prioritizing soft real-time network traffic in virtualized hosts based on xen," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015, pp. 145–156.
[25] C. L. Liu *et al.*, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, p. 46–61, Jan. 1973.
[26] M. A. El Sayed, E. S. M. Saad, R. F. Aly, and S. M. Habashy, "Energy-efficient task partitioning for real-time scheduling on multi-core platforms," *Computers*, vol. 10, no. 1, p. 10, 2021.
[27] A. Mascitti *et al.*, "Heuristic partitioning of real-time tasks on multi-processors," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2020, pp. 36–42.
[28] S. Baruah, "Partitioned edf scheduling: a closer look," *Real-Time Systems*, vol. 49, no. 6, pp. 715–729, 2013.
[29] A. Mascitti, T. Cucinotta, M. Marinoni, and L. Abeni, "Dynamic partitioned scheduling of real-time tasks on arm big. little architectures," *Journal of Systems and Software*, vol. 173, p. 110886, 2021.
[30] A. Roy *et al.*, "Energy-aware primary/backup scheduling of periodic real-time tasks on heterogeneous multicore systems," *Sustainable Computing: Informatics and Systems*, vol. 29, p. 100474, 2021.
[31] A. Wiese *et al.*, "Partitioned edf scheduling on a few types of unrelated multiprocessors," *Real-Time Systems*, vol. 49, no. 2, pp. 219–238, 2013.
[32] S. K. Baruah *et al.*, "Ilp models for the allocation of recurrent workloads upon heterogeneous multiprocessors," *Journal of Scheduling*, vol. 22, no. 2, pp. 195–209, 2019.
[33] S. K. Baruah, "Task partitioning upon heterogeneous multiprocessor platforms." in *IEEE real-time and embedded technology and applications symposium*. Citeseer, 2004, pp. 536–543.
[34] ——, "Partitioning real-time tasks among heterogeneous multiprocessors," in *International Conference on Parallel Processing, 2004. ICPP 2004.* IEEE, 2004, pp. 467–474.
[35] A. Bertout *et al.*, "Workload assignment for global real-time scheduling on unrelated multicore platforms," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, 2020.
[36] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 363–376.
[37] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.
[38] "Cloud Computing Services - Amazon Web Services (AWS)," https://aws.amazon.com/, [Online; Accessed: 2021-8-03].
[39] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien, "Real-time serverless: Enabling application performance guarantees," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 1–6.
[40] M. Elsakhawy and M. Bauer, "Faas2f: A framework for defining execution-sla in serverless computing," in *2020 IEEE Cloud Summit*. IEEE, 2020, pp. 58–65.
[41] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: Running latency sensitive serverless computations at the edge," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 239–251.
[42] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 311–327.
[43] Y. Wang, "Time-Sensitive Networking (TSN) Enabling on StarlingX," https://www.youtube.com/watch?v=DgYA5m0d87U, 2020, [Online; Accessed: 2021-6-15].
[44] "Starlingx: Open Source Edge Cloud Computing Architecture," https://www.starlingx.io/, [Online; Accessed: 2021-6-15].
[45] "Kata Containers - Open Source Container Runtime Software," https://katacontainers.io/, [Online; Accessed: 2021-6-16].
[46] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–36, 2021.
[47] L. Molnár *et al.*, "Dataplane specialization for high-performance openflow software switching," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 539–552.
[48] "Open vSwitch," https://www.openvswitch.org/, [Online; Accessed: 2021-7-06].
[49] H. Fang and R. Obermaisser, "Virtual switch for integrated real-time systems based on sdn," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019, pp. 344–351.
[50] F. Dürr, "Software TSN-Switch with Linux," https://www.frank-durr.de/?p=376, [Online; Accessed: 2021-6-16].
[51] R. Nakamura *et al.*, "Grafting sockets for fast container networking," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, 2018, pp. 15–27.
[52] "napalm-automation/napalm: Network Automation and Programmability Abstraction Layer with Multivendor support," https://github.com/napalm-automation/napalm, [Online; Accessed: 2021-6-16].
[53] S. Brosky, "Shielded cpus: real-time performance in standard linux," *Linux Journal*, vol. 2004, no. 121, p. 9, 2004.
[54] J. M. López *et al.*, "Utilization bounds for edf scheduling on real-time multiprocessor systems," *Real-Time Systems*, vol. 28, no. 1, pp. 39–68, 2004.
[55] J. K. Lenstra, D. B. Shmoys, and É. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical programming*, vol. 46, no. 1, pp. 259–271, 1990.
[56] T. H. Cormen, *Introduction to Algorithms, Second Edition*. The MIT Press, sep 2001. [Online]. Available: https://www.xarg.org/ref/a/0262032937/
[57] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
[58] G. Dósa, "The tight bound of first fit decreasing bin-packing algorithm is ffd (i)$\leq$ 11/9opt (i)+ 6/9," in *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 2007, pp. 1–11.
[59] J. van den Brand, "A deterministic linear program solver in current matrix multiplication time," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2020, pp. 259–278.
[60] M. B. Cohen, Y. T. Lee, and Z. Song, "Solving linear programs in the current matrix multiplication time," *Journal of the ACM (JACM)*, vol. 68, no. 1, pp. 1–39, 2021.
[61] F. Le Gall, "Powers of tensors and fast matrix multiplication," in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, 2014, pp. 296–303.

Márk Szalay is a Ph.D. student at Budapest University of Technology and Economics. He is a member of the High Speed Network Laboratory (http://hsnlab.hu/) in the Department of Telecommunications and Media Informatics. His main research interests include Hardware (Router/switch/NIC) design, Network programming, Software-Defined Networking (SDN) and Network Function Virtualization (NFV).

Péter Mátray is a researcher at Ericsson Research. He received his Ph.D. in 2014 from Eötvös Loránd University (ELTE). During his work at the university he was focusing on topics of large-scale active Internet measurements and the efficient management and analysis of massive measurement data sets. He joined Ericsson Research in 2012, where he has been involved in various projects related to analytics, the troubleshooting of complex cloud applications, and low-latency data sharing in the cloud.

László Toka is assistant professor at Budapest University of Technology and Economics, vice-head of HSNLab (http://hsnlab.hu/), and member of both the MTA-BME Network Softwarization and the MTA-BME Information Systems Research Groups. He obtained his Ph.D. degree from Telecom ParisTech in 2011, he worked at Ericsson Research between 2011 and 2014, then he joined the academia with research focus on software-defined networking, cloud computing and artificial intelligence.