

# AIRIC: Orchestration of Virtualized Radio Access Networks With Noisy Neighbours

Josep Xavier Salvat Lozano<sup>ID</sup>, Member, IEEE, Andres Garcia-Saavedra<sup>ID</sup>, Xi Li<sup>ID</sup>,  
and Xavier Costa Perez<sup>ID</sup>, Senior Member, IEEE

**Abstract**—Radio Access Networks virtualization (vRAN) is on its way becoming a reality driven by the new requirements in mobile networks, such as scalability and cost reduction. Unfortunately, there is no free lunch but a high price to be paid in terms of computing overhead introduced by *noisy neighbors* problem when multiple virtualized base station instances share computing platforms. In this paper, first, we thoroughly dissect the multiple sources of computing overhead in a vRAN, quantifying their different contributions to the overall performance degradation. Second, we design an *AI-driven Radio Intelligent Controller (AIRIC)* to orchestrate vRAN computing resources. AIRIC relies upon a hybrid neural network architecture combining a relation network (RN) and a deep Q-Network (DQN) such that: (i) the demand of concurrent virtual base stations is satisfied considering the overhead posed by the *noisy neighbors* problem while the operating costs of the vRAN infrastructure is minimized; and (ii) dynamically changing contexts in terms of network demand, signal-to-noise ratio (SNR) and the number of base station instances are efficiently supported. Our results show that AIRIC performs very closely to an offline optimal oracle, attaining up to 30% resource savings, and substantially outperforms existing benchmarks in service guarantees.

**Index Terms**—Open RAN, noisy neighbours problem, RAN virtualization, deep Q-learning.

## I. INTRODUCTION

RADIO Access Network (RAN) virtualization is well-recognized as a key technology to increase cost-efficiency at the very edge of next-generation mobile systems [1]. The urge to increase the density of radio access points—yet preserve or even reduce costs—has attracted the attention of industry in this direction; see, e.g., initiatives such as the O-RAN alliance [2] or Rakuten’s greenfield deployment in Japan [3]. Virtualized RANs (vRANs) are expected to

Manuscript received 7 January 2023; revised 2 June 2023; accepted 2 August 2023. Date of publication 5 December 2023; date of current version 17 January 2024. This work was supported in part by the European Commission under Grant SNS-JU-101097083 (BeGREEN) and Grant 101017109 (DAEMON), in part by MINECO/NG EU under Grant TSI-063000-2021-7, and in part by the CERCA Programme. (Corresponding author: Josep Xavier Salvat Lozano.)

Josep Xavier Salvat Lozano, Andres Garcia-Saavedra, and Xi Li are with NEC Laboratories Europe GmbH, 69115 Heidelberg, Germany (e-mail: josep.xavier.salvat@neclab.eu; andres.garcia.saavedra@neclab.eu; xi.li@neclab.eu).

Xavier Costa Perez is with NEC Laboratories Europe GmbH, 69115 Heidelberg, Germany, also with the i2CAT Foundation, 08034 Barcelona, Spain, and also with ICREA, 08010 Barcelona, Spain (e-mail: xavier.costa@ieee.org).

Color versions of one or more figures in this article are available at <https://doi.org/10.1109/JSAC.2023.3339749>.

Digital Object Identifier 10.1109/JSAC.2023.3339749

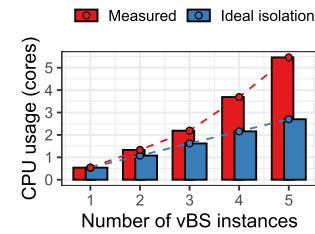


Fig. 1. vRAN per-core CPU usage with # of vBS.

import the advantages of NFV such as resource multiplexing by sharing infrastructure [4]. The idea of RAN pooling is not new: 71% of US operators indicated the intent to deploy RAN centralization by 2025 in a recent survey [5], e.g., NTT Docomo, Ericsson or AT&T are famously interested this type of technologies [6], [7], [8]; and centralization is at the forefront of O-RAN [9, §5.1.3]. However, the real-time impact of resource contention in shared RAN pooling platforms has not been studied sufficiently.

The success of Network Function Virtualization (NFV) has spurred the market to build virtual network functions (VNFs) such as firewalls, switches, VPNs, etc., that provide carrier-grade performance. However, research has shown that resource contention caused by VNFs sharing common computing infrastructure may lead to up to 40% of performance degradation compared to dedicated platforms [10], [11]. The term *noisy neighbor* problem has been coined to refer to this issue, and has motivated substantial research over the years [10], [11], [12], [13], [14]. See our review on the related work in §VI.

The virtualization of base stations (vBSs) is not alien to this issue. We confirm this with our own findings from experiments in a proof-of-concept vRAN system comprised of instances of a full-fledged 3GPP Rel.10 compliant vBS implemented with srsRAN [15]. Using Docker container techniques, we deployed a set of 10MHz vBS instances in a pool of CPU cores from an Intel core i7-7700K CPU @ 4.20GHz in a shared off-the-shelf server. The details of our experimental setup will be presented later. We then initiated bidirectional data flows, both uplink (UL) and downlink (DL), with maximum load and good wireless channel conditions between each vBS instance and a corresponding legacy user equipment (UE).

Fig. 1 depicts the relative CPU usage of the system as a function of the number of vBS instances deployed. The bars in

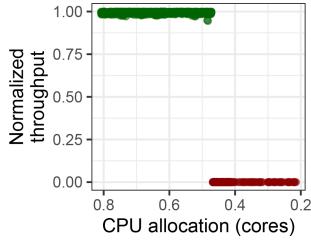


Fig. 2. Throughput vs. CPU allocation.

blue show the expected usage assuming perfect resource isolation in place. We compute these by linearly scaling up the CPU usage of a single vBS instance. The red bars show the actual CPU consumption, which unveil an exponentially-growing overhead induced by the aforementioned resource contention in imperfectly isolated computing platforms.

In the context of vRAN, exploring the gains and impact of radio network function virtualization may prove challenging to consider RAN specific characteristics. First, the vBS workload has strict time deadlines, which makes them much more sensitive to the *noisy neighbors* problem than classical VNFs such as switches or firewalls. We confirm this in Fig. 2, which shows the normalized throughput performance of one vBS for different CPU allocations (x-axis). Note that its throughput rapidly collapses upon deficit of computing resources. This occurs because physical layer (PHY) deadlines are missed, which causes that users lose synchronization with the vBS, resulting in connectivity loss [4]. This differs significantly from the cases of regular VNFs, which suffer from a smoother performance degradation upon computing resource shortages. Hence, it is an essential problem to compute required shared computing resources for vRAN deployments accounting for such impact of the *noisy neighbour* problem, which is the aim of this work.

vRANs inspired remarkable work over the last few years. In the industry, Intel FlexRAN and NVIDIA Aerial are vRAN solutions that use dedicated hardware accelerators, which are overly expensive and energy-consuming [4]. In the academia, Agora [16] proved that RAN PHY tasks can be executed in many-core general-purpose CPU platforms with carrier-grade performance, but it requires CPU cores to be *dedicated* to specific tasks (i.e., no sharing). More recently, Concordia [17] proposed an approach to share computing resources with latency-elastic applications. However, *how to share computing resources across several vBSs remains an open question*.

Although much work has studied the noisy neighbours problem on NFV workloads [10], little research has been done on the vRAN case. Nuberu [4] provides a RAN PHY processing pipeline that increases its reliability upon computing capacity fluctuations but it does not deal with the CPU allocation problem. vrAIn [18], [19] does address this problem but it does not consider the impact of the noisy neighbors problem (perfect resource isolation is assumed) and it does not support a variable number of vBSs in the system (see §V). To the best of our knowledge, we are the first to address the vRAN noisy neighbor problem on shared computing platforms (see related

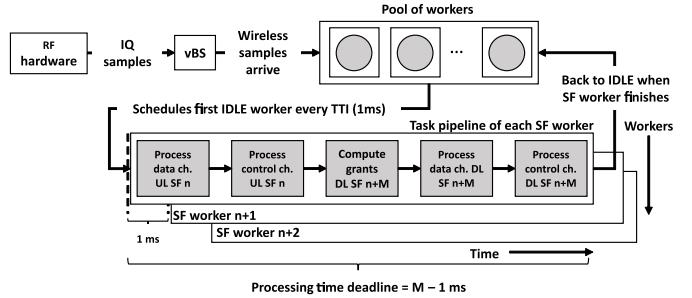


Fig. 3. Every TTI a vBS needs to spawn a new thread for its pool to process the different tasks for UL and DL.

work in §VI). More specifically, we provide the following contributions:

- In §III, we provide an in-depth analysis of the overhead incurred by multiple vBSs sharing a common CPU pool.
- In §IV, we design a data-driven model called AIRIC to optimize the allocation of computing resources in a vRAN. Compared to state-of-the art solutions, our approach learns to compensate for the overhead caused by resource contention and supports a varying number of vBS instances without requiring independent models.
- In §V, we empirically compare AIRIC with related solutions [18], [19] and with an optimal offline oracle. We show that AIRIC achieves close-to-optimal performance and over 99.9% throughput service. In contrast, previous solutions provide barely 7% savings in computing resources at a price of up to 50% throughput loss.

## II. BACKGROUND

### A. Radio Access Network Virtualization

It is well-known that the physical layer (PHY) of a vBS stack carries most of the computing heavy-lifting [20]. We next provide some background about this. Fig. 3 illustrates the operation of a Frequency Division Duplex (FDD) vBS PHY processor [4].

Every 1 ms, a vBS receives the radio samples associated with an uplink subframe  $n$ . A dispatcher selects an idle worker, which initiates a pipeline of radio processing tasks in an independent computing thread. These tasks include (i) processing the data and control channels carried by the UL subframe  $n$ , (ii) scheduling UL/DL radio grants to be transported by DL subframe  $n + M$ , (iii) processing data and control channels for DL subframe  $n + M$ , and (iv) send the modulated symbols corresponding to DL subframe  $n + M$  to the radio frontend. In 4G LTE,  $M = 4$  in respect to 3GPP constraints to provide hybrid ARQ feedback to the users, but this parameter is configurable in 5G New Radio [21].

Processing channels in a subframe consists of additional pipelines of operations, including (de)modulation of OFDM symbols or forward error coding (FEC) operations, which are compute-intensive. However, a downlink subframe has to be generated every 1 ms, and an uplink subframe has to be processed every 1 ms. To give the worker some slack to execute its job, pipeline parallelization is used. That is, a pool of  $M - 1$  workers shall be available to execute jobs. Once a worker finishes a job, it becomes idle awaiting new jobs.

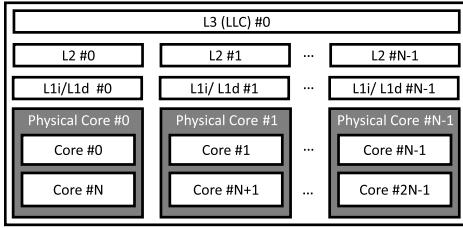


Fig. 4. General-purpose CPU platform architecture.

Among all virtualization technologies available today (virtual machines, unikernels, containers), we believe that Docker containers are the best fit to support the requirements of vRAN workloads [2]. To begin with, Docker containers support online granular resource allocation and orchestration of multiple tenants across multiple hosts. Furthermore, as opposed to virtual machines, Docker supports fast and live migration of containers, as well as easy and quick creation, upgrade, and deployment of images.

### B. General-Purpose Computing

Fig. 4 presents the CPU architecture of a general-purpose computing platform (GPP). Modern superscalar processors leverage on simultaneous multithreading (SMT) (also known as Hyper-threading in Intel CPUs), which allows a physical core to run more than one thread at a time. Thus, physical cores are seen from the operating system as two separated cores. These cores are virtual and share the same physical processor.

The cache memory is the closest and fastest memory of the CPU. It bridges the gap between RAM memory speed and CPU speed. Cache memory is usually organized in different levels regarding speed and size [22]. Level 1 (L1) cache memory is the closest and fastest memory of the system but also its capacity is the most limited. Each physical core has its dedicated L1 cache. L2 cache is bigger than L1 but slower, and it is also dedicated to each physical core. As opposed to L1, L2 is generally used for data rather than instructions. Finally, L3 cache or Last Level Cache (LLC) is the slowest cache of a CPU, and it is shared across all cores.

In a GPP, a core executing a thread loads the most used memory blocks into a cache for faster access. Then, every time a thread references a memory block that is not in a cache, the core triggers an interrupt called a “cache miss”, and looks for the data in a higher-layer memory cache (or RAM, ultimately).

### C. O-RAN Architecture

The O-RAN alliance [2] is a joint collaboration between leading industry and carrier partners in the mobile communications sector to redesign future Radio Access Network (RAN) technologies. Its main goal is to define a technical standard for RAN architecture that fosters innovation, interface openness, and reducing operational and deployment costs thanks to virtualization and general-purpose hardware.

Fig. 5 depicts the general outlook of the O-RAN architecture. O-RAN splits the BS functions into three Network Functions (NFs): (i) a Radio Unit (O-RU), (ii) a Distributed

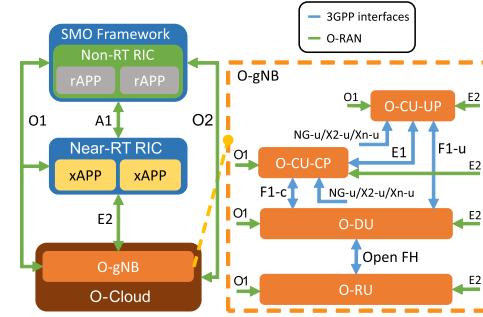


Fig. 5. O-RAN architecture.

Unit (O-DU), and (iii) a Central Unit (O-CU) [23]. The O-RU hosts low-level PHY functions, including FFT and other RF functions such as amplification or sampling. The O-DU hosts the RLC, MAC and high PHY layers, which include FEC encoding and decoding. Finally, the O-CU, which is split into two components for the user plane (UP) and control plane (CP) functions, supports the higher layer protocols as SDAP, RRC, and PDCP. Furthermore, O-RAN specifies the O-Cloud platform, which hosts virtualized NFs (VNFs) from the O-gNB.

To control and orchestrate the O-Cloud infrastructure and the O-gNB functions, O-RAN introduces two Radio Intelligent Controllers (RICs): the non-real-time RIC (non-RT RIC) and the near-real-time RIC (near-RT RIC). The Service Management and Orchestration (SMO) framework hosts the non-RT RIC, which enables control loops across large time-scales (i.e., seconds or minutes). Applications leveraging on the non-RT RIC control are called rApps. On the other hand, the near-RT RIC supports control loops on smaller time-scales (tenths of milliseconds) through applications called xApps.

The O1 Interface is a logical connection between all O-RAN components and the SMO framework. The purpose of O1 interface is to ensure the operation and management i.e. fault, configuration, accounting, performance, and security (FCAP) of the O-RAN components. The components managed via O1 include the near-RT RIC, the O-CU, the O-DU. Moreover, The near-RT RIC uses the A1 interface to receive policies from the non-RT RIC, and E2 interface to collect near-real-time information from the O-RAN components and perform fine-grained radio resource management (RRM) policies over them. Finally the SMO performs O-Cloud management and orchestration via the O2 interface.

## III. EXPERIMENTAL ANALYSIS

We first investigate the root cause of the computing overhead when multiple vBS instances share a common GPP.

### A. vRAN Testbed

To this end, we emulate a vRAN system with an off-the-shelf server and up to 10x software-defined radio (SDR) Ettus USRP B210 front-ends for both vBSs and the corresponding UEs which allows us to test up to 5 vBSs. The server provides an Intel i7-7700K CPU, with 4 physical cores and 8 virtual cores. The L1, L2, and L3 caches have 256 KiB,

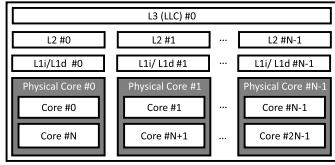


Fig. 6. Hyper-threading *vs.* no hyper-threading.

1 MiB, and 8 MiB capacity, respectively. To implement a vBS, we use a *full* 3GPP Rel.10-compliant stack from srsRAN [15] containerized with Docker, and we pair each vBS with one UE to generate downlink (DL) and uplink (UL) network load. Unless otherwise stated, the default bandwidth of each vBS is 10 MHz and we use  $N = 3$  physical cores in the experiments shown in this section. Using Docker’s API, we developed a set of custom tools to dynamically orchestrate the vRAN system and configure different parameters related to the radio and the computing settings in run-time.

### B. Hyper-Threading

Previously, we described how modern processors employ SMT to optimize resource utilization within modern processor architectures. The impact of SMT on performance varies greatly depending on the application at hand. When two threads necessitate the processor’s undivided attention, their execution can be hindered as they contend for processor access. However, if two threads engage in complementary tasks, with one requiring processor attention while the other focuses on reading and writing operations, SMT can yield significant cost-efficiency benefits by maximizing resource utilization. Fig. 6 shows the CPU utilization when we deploy different vBS at maximum traffic demand for uplink and downlink when using hyper-threading and when not using it. As we can see in the image, deactivating hyper-threading has a minimal performance improvement for less than 5 vBS. However, when deploying 5 vBSs in the system without hyper-threading, they cannot run with the maximum traffic demand as we have less computing capacity. Deactivating hyper-threading makes the platform more deterministic at the expense of having less computing capacity available. This is not surprising since the Linux CPU scheduler is aware of hyper-threading and leverages them through the scheduling domains [24].

### C. Network Isolation

Virtual networks incur substantial computing overhead. In the case of containers, the virtualization technology employed is a combination of network namespaces and virtual Ethernet pairs. With high data rates and small packet sizes, the number of operations that the host and the container must process consumes substantial CPU. This is a well-known problem reported in a plethora of literature [10], [25].

vBSs have (at least) two network interfaces: an interface with the backhaul, which connects vBSs to the mobile core (3GPP S1/Nn interfaces [26]) and another one that connects to other vBSs (3GPP X2/Xn interface [27]). For vRANs, network virtualization is not different than for traditional VNFs. Hence,

we expect that common network isolation techniques, through *network namespaces*, used in NFV behave similarly.

Fig. 7a compares the mean CPU usage of scenarios with 1 to 5 vBS instances sharing the same physical network interface for backhauling. All vBS instances are homogeneous, with dedicated frequencies and we saturate their wireless capacity in both UL and DL directions. Moreover, in an attempt to reduce other potential sources of resource conflict, in this case we allocate each vBS on dedicated CPU cores.

We test two cases: (i) isolating the network stack of individual vBSs from the host using different network namespaces (“Virtual netw.”), and (ii) allowing all vBS to use host’s networking without any namespace isolation (“Host netw.”). From the figure, we observe that the computing overhead of individual namespaces is negligible. The reason is that the aggregated network load generated by each BS is considerably smaller than the scenarios evaluated in the related literature [10], [25] (which handle over gigabit rates). Hence, network isolation cannot explain the computing toll showed in §I.

### D. Secure Computing Filters

Docker containers (and others) use, by default in most modern GPPs, a security feature called Secure Computing (Seccomp) filters [28]. Seccomp filters can control access to 300+ system calls (44 by default in Docker, which balances protection and compatibility). In the context of multi-tenant vRANs, this feature becomes of paramount importance to protect the underlying platform and mitigate potential attacks between potentially competing tenants.

Though the overhead of seccomp filters is less studied in the literature, there exist some prior work that report a computing cost associated with seccomp filters that ranges from < 10% (default seccomp profile in Docker) to almost 100% (with an overprotective scheme) [29] with conventional applications. To complement that work, we now study the impact of seccomp filters in the context of vRANs.

To this end, we deployed the same scenarios used in §III-C (using virtual network interfaces) and measured the CPU usage without seccomp filters (“seccomp off”) and with the default seccomp profile in Docker (“seccomp on”). In line with [29], we observe a rough 1.4% extra burden in CPU time for every vBS instance in the system, which adds up to 7% total with 5 vBS instances. This is a non-negligible overhead, yet it does not fully explain the large toll observed in Fig. 1.

### E. Context Switches

The natural next step is to study the impact of context switches. Thread contention in shared CPUs may lead to an increased number of context switches and, consequently, increase the total consumption of CPU resources.

To assess this, we repeat the same scenarios as before, and depict in Fig. 8 the aggregated CPU usage for a variable number of vBS instances. Like before, we allocate dedicated CPU cores (CPU pinning) to individual vBS instances in an

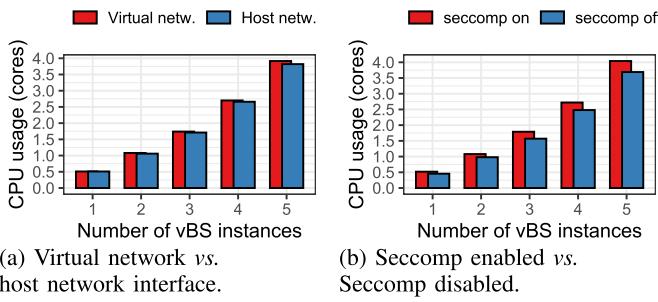


Fig. 7. 95<sup>th</sup> percentile of aggregated per-core usage of a vRAN with different number of vBS instances.

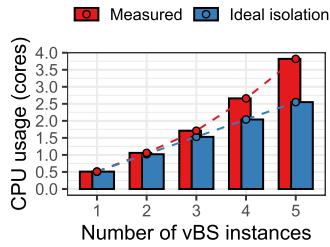


Fig. 8. 95<sup>th</sup> percentile of aggregated per-core usage with different number of vBS instances and CPU pinning.

attempt to guarantee resource isolation. In the figure, we compare our empirical result with the expected outcome with ideal isolation. Though, as expected, the impact is considerable, it only accounts to 43% of the overhead observed in Fig. 1.

To gain more insights, Fig. 9 compares the ratio of context switches experienced by an individual vBS in two different settings: (i) when each vBS is pinned to an individual CPU (as in §III-C), in Fig. 9a; and (ii) when the default CPU scheduler is free to allocate threads within the shared CPU pool (as in the experiment of §I), in Fig. 9b.

From Fig. 9a, we observe that the ratio of context switches remains very similar irrespective of the number of vBSs deployed. In this case, all the CPU contention is caused by the threads that belong to the sampled vBS. Since these are homogeneous vBSs (which implement the same amount of threads), and each of them is pinned to a dedicated CPU, the amount of contention in individual CPUs is independent of the number of vBSs deployed.

We observe a different behavior in Fig. 9b. In this case, the threads of all the vBSs compete for the same pool of CPUs. Surprisingly, when the number of vBS instances deployed in the platform is 1 or 2, the ratio of context switches is *smaller* than that when vBSs use dedicated CPUs. The reason is that the number of instances (1 or 2) is relatively smaller than the number of CPUs in the pool (6 virtual cores with  $N = 3$ ). Hence, individual threads often find less contention than in the setting used for Fig. 9a because, there, individual CPUs are dedicated to individual vBS instances but they are shared between the threads implementing the vBS (intra-vBS contention). Conversely, when the number of instances is close to the number of CPUs in the pool (4 and 5), inter-vBS thread contention dominates and the ratio of context switches noticeably overpasses that when CPUs are dedicated

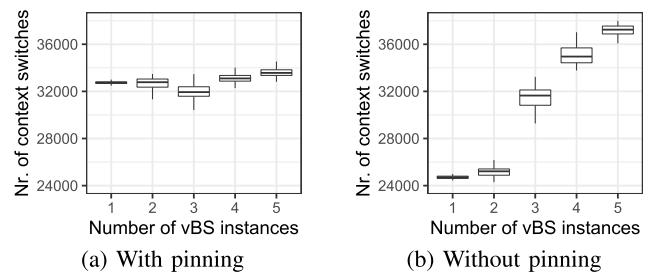


Fig. 9. Context switches per ms experienced by one vBS.

to individual vBSs. Interestingly, when we deploy 3 vBS instances, intra-vBS and inter-vBS thread contention balance out and the ratio of context switches is similar to the case when vBSs are pinned to dedicated CPUs.

With 5 vBS instances, we measure a rough 8% increase in context switches when there is no pinning with respect to using CPU pinning. Moreover, when just one vBS is deployed, there is a 24% decrease in the number of context switches that does not translate into a reduction in overall CPU time usage. Consequently, context switching cannot explain the aforementioned 43% increase in the overall CPU consumption observed in Fig. 1 with respect to Fig. 8, which lead us to the next subsection.

#### F. Cache Memory Isolation

Cache memory is a very relevant resource that is often overlooked. Although Docker provides efficient mechanisms to partition and isolate different types of resources, it does not provide features to partition cache memory resources effectively. However, cache-intensive applications sharing memory resources tend to evict each other's cache values, which increase the number of cache misses [30]. As explained in §II, cache misses cost additional CPU cycles. If data is not available in a low-level cache, a core executing a thread will trigger an interrupt signal that halts its execution until the corresponding value is finally retrieved from some higher-level memory resource. This cost in CPU cycles differ across technologies. However, we can infer its order of magnitude by observing the latency required to access different types of memory. As a reference, Table I shows the latency to access different cache levels in an Intel Skylake architecture.

To study the impact of cache contention in vRANs, we used the tool `perf` to measure the ratio of cache misses, CPU cycles and instructions required by one vBS in a system with 1-to-5 vBS instances. These measurements are summarized in Figs. 10 and 11, which show, respectively, the instructions executed per cycle (IPC), and the number of cache misses per 1000 instructions (MPKI). Both metrics show high correlation.

Fig. 10 evinces that an increasing number of vBS instances has a huge impact on computing efficiency. The red line indicates a boundary point of operation where the system process 1 instruction per cycle [33]. On the one hand, when  $IPC > 1$ , the application is instruction-bounded, i.e., only improving the efficiency of the software code can improve

TABLE I  
ACCESS AND CACHE MISS LATENCY

Memory type	Access latency [22], [31], [32]
L1 cache	4-6 cycles
L2 cache	14 cycles
L3 cache	50-70 cycles
RAM	~ 120 - 600 cycles

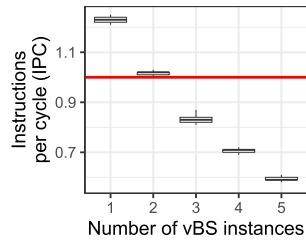


Fig. 10. Cycles per instructions (CPI) of a vBS.

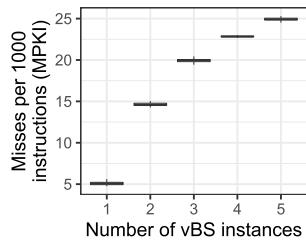


Fig. 11. Misses per instruction (MPKI) of a vBS.

the IPC performance further. On the other hand, when  $\text{IPC} < 1$ , the application is likely bounded by a bottleneck when accessing resources other than CPU, such as memory. In the case of Fig. 10 the latter occurs for a number of vBS instances larger than 2. Such a bottleneck is remarkable, allowing only 0.6 instructions per cycle when 5 vBSs are instantiated.

Conversely, Fig. 11 shows a dramatic growth of cache misses per instruction, a 500% increase with 5 vBSs with respect to 1. This, and the strong correlation between cache misses and IPC dynamics, lead us to infer that cache memory is the bottleneck in our vRAN system and, ultimately, the root cause of the anomalous CPU behavior shown in Fig. 1.

There exist mechanisms that can alleviate the impact of cache contention on CPU consumption. Perhaps the most effective approach is Intel Cache Allocation Technology (CAT) [30], which allows us to partition cache memory resources among different applications. Unfortunately, standard virtualization technologies based on cgroups (such as Docker containers) do not support such a mechanism natively. Hence, we need to find alternative strategies that allocate CPU resources to vBS instances *considering the impact of noisy neighbours problem*, which motivates our next section.

#### IV. AIRIC DESIGN

In this section, we first formalize our problem and then we describe our proposed solution, named AIRIC. AIRIC aims to

minimize the operating cost of the vRAN infrastructure (based on CPU usage). To this end, AIRIC *learns* the relationship between vBS instances, which incur resource contention in the computing platform, and network performance to optimize the allocation of computing resources in the system.

##### A. The Problem

The computing requirements of a vRAN system are hard to quantify dynamically. To begin with, the amount of CPU resources required by a single vBS instance depends on the network traffic demand on both DL and UL directions, the signal-to-noise ratio (SNR) of each wireless link and the associated Modulation Coding Scheme (MCS) used for communication, in a non-trivial manner [4], [18], [19]. Moreover, estimating the actual requirements for a set of vBS instances sharing a platform is even more challenging because the overhead introduced by computing resource contention (noisy neighbours problem) depends on the computing cores used to process each vBS workload, the amount of isolation across vBS instances, and the maximum computing capacity available.

On the one hand, over-dimensioning the allocation of computing resources incurs high infrastructure costs as many computing cores might not be needed when running a small number of vBS instances or when the aggregated load is low, and the electricity bill associated with unneeded active cores can be substantial. On the other hand, pooling a reduced number of cores across many instances (i.e., forcing vBSs to share) may lead to throughput loss because heavy resource contention leads to severe computing overheads. As we demonstrated in §I, a shortage of computing resources (due to the influence of the noisy neighbors problem) may cause that the users associated with vBSs in the system lose synchronization, induce a high number of radio link errors, and cause very high end-to-end latency and jitter.

Moreover, though pinning vBS workloads to specific CPU cores provides better isolation and performance determinism, as shown before, it requires activating a larger pool of CPU cores, which incurs higher energy costs. Hence, our approach is to let all the vBS instances fairly share a pool of CPU cores, using a standard scheduler, and determine dynamically the smallest set of active CPU cores in the pool at every time step to minimize energy costs. The key novelty in our approach is that we do so in a reliably manner, accounting for the costs of sharing, as dissected earlier. As we show later in §V, ignoring such cost has dramatic consequences on network performance.

##### B. System Model

We consider an O-RAN cloud computing platform (O-Cloud) providing computing resources for multiple vBS instances deployed therein, i.e., each vBS instance shares the same pool of computing resources. We also consider an agent in charge of (i) observing the *context* associated with each vBS, and (ii) devising which computing cores need to be active in the pool to serve the demand of each vBS, which process uplink and downlink traffic. As shown in Fig. 12,

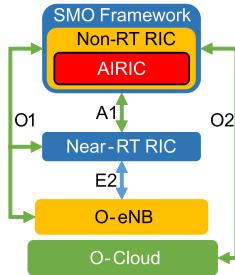


Fig. 12. AIRIC within O-RAN.

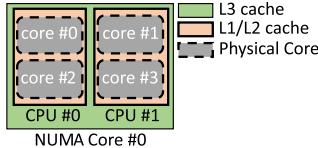


Fig. 13. Toy GPP.

following O-RAN's specification, our agent is hosted by the system's Service Management and Orchestration (SMO), and takes decisions in discrete time intervals  $t \in \mathbb{N}$ , which we call *decision intervals* and are in the range of several seconds to minutes following O-RAN's specification for the Non-Real-Time RAN Intelligent Controller (Non-RT RIC).

Our agent employs an O-RAN-compliant monitoring system that gathers metrics from the various O-RAN components (such as O-RU, O-DU, and O-CU) and measurements from the O-Cloud platform (i.e. infrastructure metrics). The near-RT RIC uses the E2 interfaces to periodically receive different radio metrics from the components deployed in the O-Cloud platform [34]. Afterward, the near-RT RIC passes the data using the O1 interface to the non-RT RIC. On the other hand, to gather metrics from the O-Cloud platform, the agent sets up performance management (PM) jobs that collect different infrastructure metrics (i.e. computing usage, energy consumption) using the O2 interface [34]. Finally, to enforce the different computing policies that our agent computes, it uses the O2 interface to pass those policies to the O-Cloud platform. Fig. 12, depicts how our agent integrates into the ORAN architecture.

Given the hard-to-model nature of the noisy neighbour problem, we advocate for reinforcement learning (RL) to design our agent. In this way, the agent observes the context and takes an action at the beginning of each decision interval, and then receives a reward at the end of the decision interval. The learning agent stores 3-tuple samples comprised of the context, actions, and the associated rewards at every interval, and uses these experiences to learn and improve the obtained rewards over time. Note that while the admission control problem is out of the scope of this paper, we do support a number of active vBS instances that may vary over time. To the best of our knowledge, this is the first solution that optimally allocates computing resources in a vRAN system accounting for the overhead of the noisy neighbours problem and a dynamically changing number of vBS instances in the system.

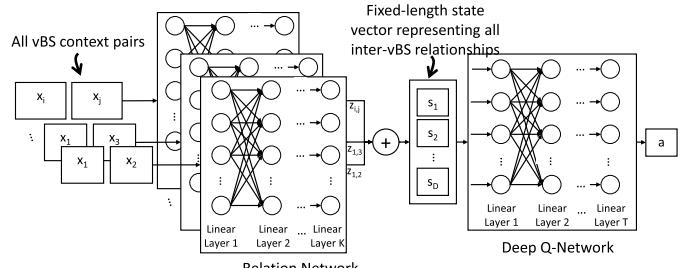


Fig. 14. AIRIC machine learning architecture.

### C. Optimization Framework

A variable number of vBS instances imply that the dimensionality of the context information also varies over time. This is particularly challenging to support with standard RL solutions. To address this, we augment a classical Deep Q-Network (DQN) approach [35] with a Relation Network (RN) mechanism [36] as shown in Fig. 14.

The basic idea of an RL agent is to learn an optimal policy  $\pi$  by interacting with an environment  $\mathcal{E}$  in discrete time intervals  $t \in \{1, 2, \dots, T\}$ . Every interval, an agent observes a state (or context)  $\vec{s}^{(t)}$ , selects an action  $a^{(t)}$  and receives a reward  $r^{(t)}$  at the end of the time step. A policy  $\pi$  is a distribution of actions over the different states, which captures the *goodness* of the state-action pair  $(\vec{s}^{(t)}, a^{(t)})$ . Once the reward  $r^{(t)}$  is measured, the system transitions to state  $\vec{s}^{(t+1)}$ . After  $T$  intervals,  $\mathcal{E}$  reaches its terminal state and the agent refines its policy  $\pi$  using past observations  $\{\{\vec{s}^{(1)}, a^{(1)}, r^{(1)}\}, \dots, \{\vec{s}^{(T-1)}, a^{(T-1)}, r^{(T-1)}\}\}$ . The goal is to maximize the total discounted reward  $R^{(t)} := r^{(t)} + \sum_{t'=t+1}^T \gamma^{t'} r^{(t')}$ .

Most RLs approximate value functions that estimate the importance of actions given a state  $\vec{s}$ . One of the those value functions is  $Q^*(\vec{s}, a) := \max_{\pi} \mathbb{E}[R^{(t)} | \vec{s}^{(t)} = \vec{s}, a^{(t)} = a]$ , which represents the maximum expected return given an action-state pair under the policy  $\pi$ . The optimal  $Q^*$ -value function follows the Bellman Optimality Equation, which provides  $Q^*(\vec{s}^{(t)}, a^{(t)})$  in terms of  $Q^*(\vec{s}^{(t+1)}, a^{(t+1)})$ :

$$Q^*(\vec{s}, a) = \mathbb{E} \left[ r^{(t)} + \gamma \max_{a^{(t+1)}} Q^*(\vec{s}^{(t+1)}, a^{(t+1)}) | \vec{s}^{(t)} = \vec{s}, a^{(t)} = a \right]$$

Using the Bellman Optimality Equation, we can find  $Q^*(\vec{s}, a)$  iteratively [37]. In this paper, we have used neural networks to approximate the optimal  $Q^*(\vec{s}, a)$ , which is called Deep Q-Network (DQN) [35]. In particular, given the large timescale of the Non-RT RIC, the action taken at one interval  $a^{(t)}$  has little impact on the next state  $\vec{s}^{(t+1)}$  and therefore it is enough to maximize instantaneous reward. Hence, to expedite convergence, we simplify our RL setting into a contextual bandit problem by setting  $\gamma = 0$  and  $T = 1$ .

We next describe our design for the learning agent's context (states), actions, and reward function.

1) *Context:* In line with the related literature [18], [19], [38], [39], we use the next metrics to describe the state:

- **Chanel quality:** We use the mean UL SNR observed by each vBS in the last interval, which allows our agent

to infer their UL wireless capacity, and the mean DL channel quality indicator (CQI) to do the same for the DL.

- **Network demand:** The network demand of a vBS is the amount of UE buffered data for both UL and DL during the last decision interval.

We represent DL and UL channel quality for a vBS instance  $i$  observed in interval  $t$  as  $\sigma_{\text{DL},i}^{(t)}$  and  $\sigma_{\text{UL},i}^{(t)}$ . Furthermore, we let  $d_{\text{DL},i}^{(t)}$  and  $d_{\text{UL},i}^{(t)}$  denote its DL and UL network demand, respectively. We also assume a known mapping between channel quality and MCS:  $g_{\text{DL}}(\sigma_{\text{DL},i})$  for DL,  $g_{\text{UL}}(\sigma_{\text{UL},i})$  for UL, which is a mild assumption. Because the channel quality bounds the highest MCS, we can estimate the mean number of radio Resource Blocks (RBs) that each vBS can use in both directions given a mean MCS and network demand. This can be estimated using the 3GPP specifications [40]. In this way, we can state the demand for radio resources (RBs) rather than relying only on the past utilization of Radio Blocks, which may differ. Consequently, we denote the number of RBs used for DL and UL for vBS  $i$  as  $p_i^{\text{DL}}$  and  $p_i^{\text{UL}}$ , respectively. Using the number of RBs and network demand, we define the context of vBS  $i$  as

$$\vec{x}_i^{(t)} := (p_{\text{DL},i}^{(t)}, d_{\text{DL},i}^{(t)}, p_{\text{UL},i}^{(t)}, d_{\text{UL},i}^{(t)})$$

The design of  $\vec{x}_i$  is motivated by the convenience of expressive features and minimal dimensionality and follows the state of the art [18], [19], [38], [39]. The challenge now is to encode the context information  $\{\vec{x}_i\}$  for all vBS instances  $i$  in a state vector  $\vec{s}$  with *fixed* dimensionality  $D$ , which is required by the DQN model, in scenarios with a variable number of vBS instances over time. As shown in Fig. 14, we address this with a Relation Network (RN) [35].

2) *Relation Network:* As the number of vBSs that AIRIC has to allocate CPU resources for in a particular time interval might be different than in past intervals, the context length changes depending on the number of vBS instances. Rather than building other agents for each of the different numbers of vBS cases or padding the various possible contexts to match a fixed context length, we opted to solve the problem using a Relation network. A RN can encode the relationship between the context associated to all vBS instances into a fixed-length state vector  $\vec{s}$ . To this end, the RN operates along all possible pairs of objects (context of vBS instances) to capture such hidden relations with a multi-layered perceptron (MLP) model. Assuming a maximum number of vBS instances supported in the system equal to  $M$ , then we have the following possible pairs of context vectors:

$$\mathcal{X} := \{(\vec{x}_1, \vec{x}_2), (\vec{x}_1, \vec{x}_3), \dots, (\vec{x}_{M-1}, \vec{x}_M)\}$$

Since the maximum amount of vBS instances at any given moment is bounded, then  $|\mathcal{X}|$  is also bounded and fixed over time. The RN ingests sequentially each pair  $(\vec{x}_i, \vec{x}_j) \in \mathcal{X}$  of possible unpermuted context combinations, and generates an output vector  $\vec{z}_{i,j}$  with cardinality  $D$ . Once all  $\binom{N}{2}$  permutation vectors  $\vec{z}_{i,j}$  are computed by the RN, which is done sequentially, we create an encoded state vector  $\vec{s}$  by aggregating all output vectors, i.e.,  $\vec{s} = \sum_{i,j} \vec{z}_{i,j}$ . In

this way, we force order permutation invariance, which is a critical requirement of our problem, i.e., as the RN learns about different latent relations across vBS instances (objects), these learned relations remain invariant regardless the order of the input pair relations. Importantly, our RN not only helps to support variable number of vBS instances over time, it also provides the DQN model with state information that represents better the relations between them, *which is very helpful to capture the impact of the noisy neighbours problem in a state dimension-fixed representation*. To this end, we train the RN network jointly with the DQN model as we explain later.

3) *Actions:* Given state  $\vec{s}^{(t)}$ , our agent shall *activate* the appropriate set of CPU cores, described with an *activation vector*  $\vec{v}$  wherein each element corresponds to the CPU core index that shall be activated. Then, all the vBS instances will fairly share the pool of CPU cores in  $\vec{v}$ . By avoiding pinning vBS workloads into specific cores, we aim at maximizing resource multiplexing and, consequently, at reducing the overall usage of computing resources. To ensure quick convergence, we need to preserve a low action space dimensionality. To address this we resolve our action into two steps. In step 1, our RL agent decides the *total* number of CPU cores that shall be activated to *guarantee service*. Thus, the set of actions  $A$  is  $A = \{1, 2, \dots, 2N\}$ , where  $N$  is the total number of physical cores available. Then, in step 2, we implement a *deterministic* rule  $\rho(a)$  to *minimize infrastructure cost*. That is,  $\rho: A \rightarrow \mathcal{V}_a, a \mapsto \vec{v}$ , where  $\mathcal{V}_a$  is a set containing all possible activation vectors such that  $a = |\vec{v}|$ . Because  $\rho$  is a pre-determined rule to minimize cost, the agent can learn its policy  $\pi$  to guarantee service given  $\rho$  as part of the environment  $\mathcal{E}$ .

See, e.g., the GPP of Fig. 13 with  $N = 2$ . If  $a = 1$  then  $\mathcal{V}_{a=1} = \{(0), (1), (2), (3)\}$  all the activation vectors in  $\mathcal{V}_{a=1}$  are equivalent and any  $\vec{v} \in \mathcal{V}_{a=1}$  could be chosen trivially. However, this is not necessarily the case for other actions  $a$  because, as we explained before, modern processors leverage multi-processing CPUs, being two virtual cores for each physical CPU the most common case. For instance, for  $a = 2$  (and the same GPP with  $N = 2$ ), the set of possible activation vectors is  $\mathcal{V}_{a=2} = \{(0, 2), (1, 3), (0, 1), (0, 3), (1, 2), (1, 3)\}$ . Though many of the vectors in  $\mathcal{V}_{a=2}$  are equivalent, others are not. Subset  $\hat{\mathcal{V}}_{1,a=2} = \{(0, 2), (1, 3)\} \subset \mathcal{V}_{a=2}$  contains equivalent activation vectors; and so are the activation vectors in  $\hat{\mathcal{V}}_{2,a=2} = \{(2, 3), (0, 1), (0, 3), (1, 2)\} \subset \mathcal{V}_{a=2}$ . But any  $\vec{v}_1 \in \hat{\mathcal{V}}_{1,a=2}$  and any  $\vec{v}_2 \in \hat{\mathcal{V}}_{2,a=2}$  are *not* equivalent. On the one hand, any  $\vec{v}_1 \in \hat{\mathcal{V}}_{1,a=2}$  incurs more cache contention than any  $\vec{v}_2 \in \hat{\mathcal{V}}_{2,a=2}$  because all the cores in  $\vec{v}_1$  share the same physical CPU (see Fig. 13). On the other hand, any  $\vec{v}_2 \in \hat{\mathcal{V}}_{2,a=2}$  is more costly than any  $\vec{v}_1 \in \hat{\mathcal{V}}_{1,a=2}$  because  $\vec{v}_1$  allows turning off more physical CPUs, e.g., if  $\vec{v}_1 = (0, 2)$  CPU 1 can be turned off (see Fig. 13). Fig. 15 illustrates an example of the operation of our algorithm during three time steps. Importantly, give a pool of activated CPU cores, all vBS instances will fairly use those cores using a standard scheduler.

In the assumption that, given any static mapping  $\rho$ , policy  $\pi$  will provide an appropriate cardinality for the activation vector

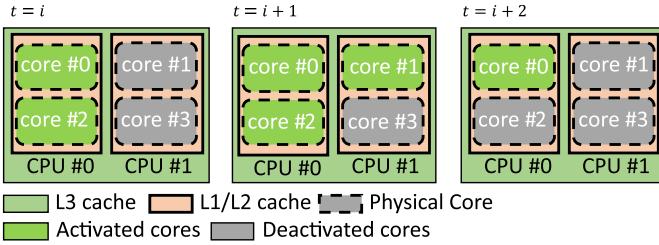


Fig. 15. AIRIC actions timeline.

to guarantee network service ( $a = |\vec{v}|$ ), we just need to design  $\rho$  aiming to minimize the amount of infrastructure (physical CPUs) that has to be activated given  $a$ . Consequently, we propose the following simple rule. Let  $k(\vec{v}) \in \{1, 2, \dots, N\}$  denote the number of physical CPUs that contain at least one virtual core activated in  $\vec{v}$ . Then, given a set  $\mathcal{V}_a$  with all possible activation vectors for action  $a$ , we define the *ordered superset*  $\mathcal{W}_a := \langle \hat{\mathcal{V}}_{1,a}, \dots, \hat{\mathcal{V}}_{N,a} \rangle$ , where  $\mathcal{V}_{i,a} = \{ \vec{v} | k(\vec{v}) = i, \vec{v} \in \mathcal{V}_a \}$ . In the example above, with  $a = 2$  and  $N = 2$ ,  $\mathcal{W}_a = \{ \hat{\mathcal{V}}_{1,a=2}, \hat{\mathcal{V}}_{2,a=2} \}$ . Note that  $\hat{\mathcal{V}}_{i,a} = \emptyset$  for some  $i$ . For instance, in our toy example with  $N = 2$ ,  $\hat{\mathcal{V}}_{1,a=3} = \emptyset$  for  $a = 3$ . Hence, we let  $\rho(a) = \vec{v} \in \hat{\mathcal{V}}_{m,a}$  such that  $m := \arg \min_i \{i | \hat{\mathcal{V}}_{i,a} \neq \emptyset\}$ .

4) *Reward*: Our goal is to meet the traffic demand of all the vBS deployed in the system over time with minimum physical infrastructure (to save costs by turning off CPUs). Assuming a pool with  $N$  physical CPUs and  $2N$  virtual cores, where cores  $j$  and  $j+N$  belong to the same physical CPU  $\forall j < N$ , we let  $z(j) \in \{0, \dots, 2N-1\}$  denote the *sibling* virtual core  $i$  given input virtual core  $j$ . A sibling core is that that uses the same physical CPU. For instance, in the toy GPP of Fig. 13, with  $N = 2$  physical CPUs and 4 cores,  $z(0) = 2$  and  $z(2) = 0$ .

Following the related literature [41], [42], we codify the cost associated to an activation vector  $\vec{v}$  using a linear model. Let us first denote  $c_j^{(t)} \in [0, 1]$ , as the relative usage of computing core  $j$  during interval  $t$ . If  $j \notin \vec{v}^{(t)}$ , then  $c_j^{(t)} = 0$ ; otherwise,  $c_j^{(t)}$  is empirically measured. Then, we let  $E_j^{(t)}$  model the (energy-related) cost associated with computing core  $j \in \{0, 1, \dots, 2N-1\}$  as follows:

$$E_j^{(t)} := \begin{cases} \alpha_1 + \beta \cdot c_j^{(t)} & \text{if } c_j^{(t)} > 0 \\ \alpha_2 & \text{if } c_j^{(t)} = 0 \text{ and } c_{z(j)}^{(t)} > 0 \\ \alpha_3 & \text{if } c_j^{(t)} = 0 \text{ and } c_{z(j)}^{(t)} = 0 \end{cases} \quad (1)$$

where  $\alpha_1 > \alpha_2 > \alpha_3$ . Intuitively,  $\alpha_i$  models the bias cost of a core, which is different depending on the activation state of core  $j$  and its sibling. We choose  $\alpha_i$  and  $\beta$  so that  $0 \leq E_j \leq 1$ .

We now let  $\tau_{DL,i}^{(t)}$  and  $\tau_{UL,i}^{(t)}$  denote the DL/UL throughput experienced by vBS  $i$  during interval  $t$ , and then formalize our reward function as:

$$r^{(t)} := \begin{cases} -1, & \text{if } \tau_{DL,i}^{(t)} < d_{DL,i}^{(t)} \text{ for any } i \\ -1, & \text{if } \tau_{UL,i}^{(t)} < d_{UL,i}^{(t)} \text{ for any } i \\ \frac{1}{2N} \sum_{j=0}^{2N-1} -E_j, & \text{otherwise} \end{cases} \quad (2)$$

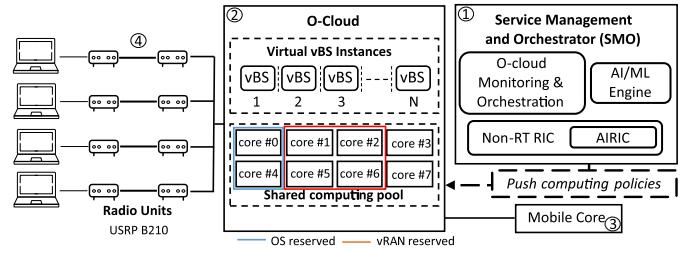


Fig. 16. Conceptual design of the evaluation testbed.

5) *Training*: As explained above, the goal is to train a policy to approximate an optimal action-value function  $Q^*$ . Our policy  $\pi$  is implemented by the structure of RN+DQN introduced above and, hence, we shall optimize the weights  $\vec{\Theta} := (\vec{\theta}_1, \vec{\theta}_2)$  of the combined neural networks to estimate the Q-value function  $Q(s, a; \theta) \approx Q^*(s, a)$ . To this end, we use a Smooth L1-loss function [43].

$$L^{(t)}(\Theta_i) := \begin{cases} \frac{1}{2} \frac{x^2}{1} & \text{if } |x| < 1 \\ |x| - \frac{1}{2} \cdot 1 & \text{otherwise} \end{cases} \quad (3)$$

where  $x = \mathbb{E}_{(s, a, r, s') \sim \rho} [(y_i - Q(s, a; \Theta_i))]$  and  $y_i = r + \gamma \max_{a(t+1)} Q(s', a(t+1); \Theta_{i-1})$ .  $\rho$  is a replay buffer from where we sample  $(s, a, r, s')$ ,  $y_i$  is the temporal difference target, and  $y_i - Q$  is the temporal difference error. We use a target network to stabilize the training process, that is, the learning agent uses a different target network with fixed weights that are used to compute the loss function used in turn to train the primary  $Q$ -network. It is crucial to stress that the target network's parameters are periodically synchronized with those of the primary  $Q$ -network rather than being trained. The primary  $Q$ -network is trained using the target network's  $Q$  values in an effort to increase the training's stability. Finally, we use a standard  $\epsilon$ -greedy approach for exploration.

## V. PERFORMANCE EVALUATION

We have built an O-RAN-compliant experimental testbed to evaluate AIRIC. The testbed comprises different hosts, which contain the components of an O-RAN deployment and the ones to provide network connectivity to different connected UEs. Fig. 16 depicts conceptually the testbed that we have built. First, this testbed has a host, which deploys the SMO and contains the non-RT RIC where we deploy AIRIC (①). Second, it has a separate host that hosts the O-Cloud platform where different O-eNB instances can be deployed and also comprises the near-RT RIC (②). To implement the orchestration and management functions of the O-Cloud platform provided by the SMO, we have opted to implement the O-eNBs deployed in the O-Cloud platform, containerizing srsRAN using Docker. Thus, we use Docker API capabilities to orchestrate and manage containers to implement a minimal O2 interface. In addition, we used a metrics agent as Telegraf to implement the performance monitoring jobs, which allowed us to gather metrics from the O-Cloud platform. Rather than using a commercial orchestrator such as Kubernetes or Docker

swarm, we implemented our minimal orchestrator for performance and flexibility. Moreover, we have also implemented minimal O1 and E2 interfaces to allocate resources on the vBSs deployed. Our testbed also includes a host, which contains the EPC to provide connectivity to the different UEs attached to each vBS (3). As the vBSs are containerized using Docker we have isolated the networking from each one another.

The O-Cloud host comprises an Intel i7-7700K GPP with 4 physical CPUs. We use Ubuntu 20.04.5 LTS with kernel 5.13.19. We reserve 1 physical CPU (2 virtual cores) for the OS and custom scripts to manage the experiments, interact with Docker API, and collect data, i.e., we emulate a small GPP vRAN platform with  $N = 2$  physical CPUs and 4 virtual cores (as in Fig. 13). The testbed also integrates 4 USRP SDR boards to support up to 4 vBS (and the corresponding UEs to generate network load) (4). To generate uplink and downlink flows, we use mgen<sup>1</sup> to initiate a flow from/to the UE to/from the EPC. Given the constrained computing capacity of our testbed, we set the bandwidth of each vBS to 10 MHz. We have generated 60k context-action-reward data samples, evenly split for scenarios with 2, 3 and 4 vBS instances operating concurrently. We shuffled and split the dataset into a training and a testing set of 40k and 20k samples, respectively.<sup>2</sup>

We have implemented AIRIC using PyTorch.<sup>3</sup> On the one hand, the RN has one hidden layer and the same number of neurons than the output layer, 128. On the other hand, the DQN has one hidden layer with 256 neurons. The initial parameters of the neural networks are initialized from an uniform distribution. We also use the ReLu activation function, and a normalization layer [44] in between hidden layers. For the  $\epsilon$ -greedy mechanism, we use a decay factor equal to 60% of the size of the training set. We also use a replay buffer with 20k samples and batches of 128 samples. Finally, we used Adam [45] as our optimizer. These implementation choices are intended to stabilize training based on [44] and [46].

### A. Convergence Evaluation

We start evaluating convergence. Fig. 17 shows the normalized reward of AIRIC over training iterations. The UL/DL load and SNR generated in both plots are chosen uniformly at random. However, while the number of vBS instances is also random (between 2 and 4) in Fig. 17a, they arrive sequentially in Fig. 17b. In the former case, the reward converges to 0.95 in less than 5k iterations. In the latter case, there are expected bumps when new vBSs arrive but these are small, within 5%. Hence, we conclude that the RN in AIRIC learns correctly the relationship across vBSs and how to use its experience to quickly reach close-to-optimal performance.

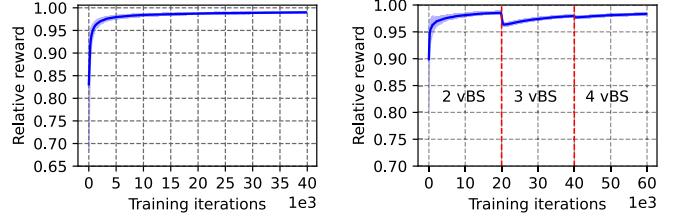
### B. Inference Time

In order to assess whether AIRIC is suitable for running in a non-RT RIC controller, we measured the inference time

<sup>1</sup><https://github.com/USNavalResearchLaboratory/mgen>

<sup>2</sup>Our dataset will be publicly available upon publication.

<sup>3</sup><http://www.pytorch.org>



(a) Randomized contexts      (b) Sequential number of vBSs

Fig. 17. AIRIC convergence evaluation.

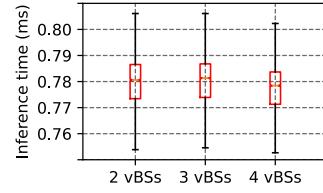


Fig. 18. AIRIC's inference time.

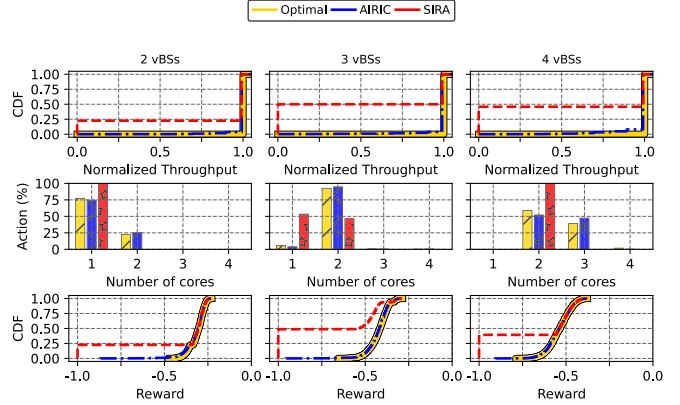


Fig. 19. Performance benchmarking.

of our approach for the different number of vBS cases. The results, depicted in Fig. 18, shows inference times lower than 1 millisecond (ms) for all cases, which is well below the control-loop cycle of a RIC controller and validates AIRIC to operate therein appropriately.

### C. Performance Benchmark

To better understand the effectiveness of our solution, we now compare AIRIC against a Single Instance Resource Allocation (SIRA) approach. SIRA is purposely designed to orchestrate optimal resources across vBS instances *under the assumption of full computing isolation between instances*. Consequently, SIRA represents upper bounds attainable by existing works on vRAN CPU orchestration such as [19] and [47].

To evaluate AIRIC, at every interval we choose uniformly at random the number of vBS instances, their DL/UL load and their DL/UL SNR, and use both approaches (AIRIC and SIRA) to optimize the allocation of computing resources dynamically. In the case of SIRA, we use different (previously trained) models depending on the number of instances. For comparison, we also depict the performance of an oracle, labelled as “Optimal”, that finds the optimal action offline by exhaustive search.

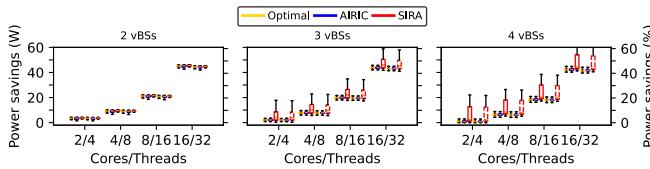


Fig. 20. Power consumption savings.

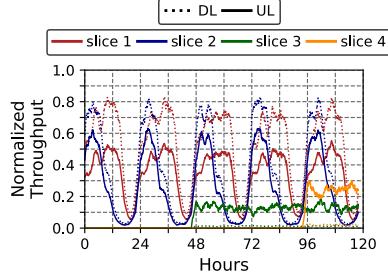


Fig. 21. Realistic load traces.

Fig. 19 depicts the distribution of the normalized aggregate throughput performance of the system (top), the CPU assignments (middle), and the distribution of the reward achieved (bottom), for all the approaches conditioned to the presence of 2 (left), 3 (middle) and 4 (right) vBS instances. Conversely, Fig. 20 depicts the absolute (left y-axis) and relative (right y-axis) power consumption savings achieved by all three approaches. These savings are in comparison to the power consumed when the default Linux scheduler manages all available CPU cores in the system, as indicated on the x-axis. The box plots represent the 25th and 75th percentiles (edges of the box), the median (line within the box), and the 5th–95th percentiles (error bars). We make three observations: The first observation is that AIRIC provides substantial savings, comparable to the optimal benchmark. Perhaps surprisingly, SIRA shows mildly higher savings in some cases, which leads to our second observation: the savings provided by SIRA come at a huge price in throughput performance, as shown by Fig. 19. This is worse for denser scenarios: with 4 vBSs, SIRA barely saves 7% computing resources more than AIRIC in average but incurs 50% throughput loss in exchange. This is due to the fact that SIRA ignores the additional computing overhead caused by the noisy neighbour problem and often under-allocates resources, leading to PHY violations and throughput loss. The final observation is that AIRIC provides a throughput performance that is remarkably close to that of ‘‘Optimal’’. Moreover, Fig. 19 (bottom) confirms that the reward distribution attained by AIRIC is very close to the provided by the optimal oracle. These observations validate our design.

#### D. Realistic Context Traces

We finally test AIRIC with realistic context dynamics. To this end, we have generated context profiles for 4 different vBS instances, implementing *network slices* with different context profiles, during 5 straight days. Fig. 21 shows the time evolution of both DL and UL network load for these 4 traces. Slice 1 emulates the behavior of one eMBB vBS in the city

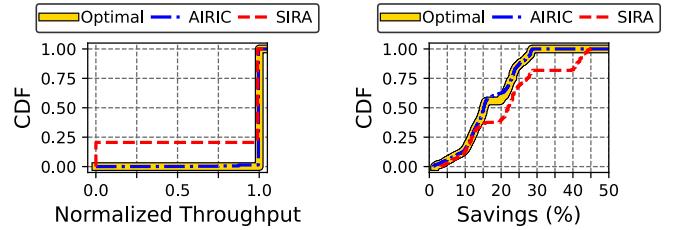


Fig. 22. Dynamic context profiles based on realistic traces.

center, with common diurnal load patterns. Slice 2 emulates a vBS serving an office building, with a peak load during office hours (9h - 17h). Both context dynamics are adapted from those in [38]. Slice 3 and 4, in turn, emulate IoT-serving vBSs with constant loads when they are operative.

Fig. 22 depicts the distribution of the throughput performance (left) and the computing resource savings (right) of AIRIC, SIRA and the optimal oracle. Like before, SIRA provides around 5% higher CPU savings in average but incurs almost 25% throughput loss over the 5 days as a consequence. Conversely, AIRIC performs very closely to the oracle, with no throughput loss and around 17% overall computing resource savings, which validates AIRIC for realistic scenarios.

## VI. RELATED WORK

### A. vRAN Orchestration

There has been quite a number of pioneering work on the vRAN orchestration that embraces and builds upon the Open RAN paradigm to provide intelligent solutions on resource allocation for the deployment of vBSs over commercial off-the-shelf computing platform (e.g., [18], [38], [39], [48]) and provide energy-aware solutions (e.g., [4]) to optimize the energy consumption of underlying computing resources. For instance, [48] presents the implementation of a vBS capable of supporting URLLC slices. In the spectrum of computing resource allocation problems, the work of [38] introduced a Bayesian learning model to optimize radio policies subject to hard power consumption constraints. EdgeBol [39] proposed a non-real-time learning algorithm to optimize radio policies and non-radio service parameters jointly, and Concordia [17] addressed sharing computing resources with latency-elastic applications.

RAN virtualization also enables sharing computing resources to reduce costs. Making a decisive step forward towards cost-effective implementation of virtual and Open RAN, vrAIn [18] was the first work to jointly optimize the CPU allocation and radio policies for a given number vBSs deployment. More recently, [47] provided a solution to allocate computing resources among a vBS instance and a vertical service. They are considered as the most pioneer and relevant benchmark related to our work. But neither of the work look into and explore the *noisy neighbor* problems caused by imperfect resource isolation over computing resources that are shared among virtual base stations, and no solution exist so far on computing required shared computing resources accounting for the impact of *noisy neighbors* problem, which is however significant on the vRAN performance, as pointed out in §I and as analyzed in §III. Moreover, as shown in §V, this type of

solutions requires independently-trained models depending on the total number of vBS deployed in the system. In contrast to all the prior work which does not support variable number of vBS instances, our approach learns the relationship between vBS instances and adapts naturally to different amount of instances over time.

The *noisy neighbor* problem in shared computing and networking environments has been extensively studied for cloud or container-based systems, but to the best of our knowledge, our work is the first to address this problem for vRAN shared computing platforms. In the following, we provide a sample of the most relevant contributions concerning isolation techniques, which are related to our analysis in §III.

### B. Network Isolation

Noisy neighbor problems can be due to imperfect network traffic isolation. Different enforcement schemes have been proposed to ensure a high degree of traffic isolation among consolidated NFs, for instance, [25] accounted for the time spent in the networking stack on behalf of a container, and [10] enhanced the cache isolation with careful sizing of I/O buffers, and [49] designed *NetBricks* framework which embraced the zero-copy software isolation ideas.

### C. Secure Computing Filters

*Seccomp* [28] related work is mainly found in the computer security realm to harden security against attacks. Reference [11] proposed a reliable method to generate custom Seccomp profiles for arbitrary containerized applications to improve container security. Reference [29] proposed *Draco* to address the lengthy rule-based checking programs against system calls and their arguments which lead to substantial execution overhead. And [50] proposed *Chestnut*, an automated approach for generating strict syscall filters of Seccomp with lower requirements and more restrictions.

### D. CPU Isolation

Most work in this area is focused on advancing the CPU scheduling to prevent overheads caused by inter-core communication and context switching. For instance [51] developed a network packet processing platform built on top of the KVM platform and Intel DPDK library to support high-speed inter-VM communication through the scheduling VMs across different CPU cores. Besides, there are also some amount of work on exploring mapping of kernel thread partitioning techniques to CPU/GPU cores (e.g., [52], [53]).

### E. Cache Memory Isolation

One of the main causes of noisy neighbor problems is cache memory sharing, and more specifically the last-level cache (LLC). To address this problem, several works have proposed optimizing LLC partitioning and adopting Cache Allocation Technology (CAT) [54], [55]. In general, there are many different approaches to implement cache memory isolation, either by software (e.g., [56]) based on page coloring technique or hardware (e.g., [57]) cache partitioning, or a combination of both (e.g., [58]).

## VII. CONCLUSION

Contention for computing resources can jeopardize the performance and costs of virtualized radio access networks at scale as the number of base stations *sharing* a computing platform grows. In our work, we have untangled the main sources for the increasing *noisy neighbor* problem in vRANs (*namespaces, context switches, security filters, cache contention*) and quantified their relative impact towards the overall computing overhead. In order to address the identified noisy neighbor problem in vRANs, we have designed AIRIC, which can adapt to varying contexts reconfiguring computing platforms dynamically and achieving nearly the performance of an offline optimal oracle. Our results show that AIRIC correctly dimensions the pool of computing cores and prevents vBSs from throughput collapse by accurately predicting the noisy neighbours problem. AIRIC leverages on a hybrid learning architecture comprising a Relation (RN) and a Deep Q-Network (DQN) to predict the best hardware configurations over time and counter the vRAN computing platforms *sharing* negative effects; attaining over 99.9% service availability and up to 30% resource savings.

## REFERENCES

- [1] Samsung. (2019). *Virtualized Radio Access Network: Architecture, Key Technologies and Benefits*. [Online]. Available: [https://images.samsung.com/is/content/samsung/p5/global/business/networks/insights/white-paper/virtualized-radio-access-network/white-paper\\_virtualized-radio-access-network.pdf](https://images.samsung.com/is/content/samsung/p5/global/business/networks/insights/white-paper/virtualized-radio-access-network/white-paper_virtualized-radio-access-network.pdf)
- [2] A. Garcia-Saavedra and X. Costa-Pérez, "O-RAN: Disrupting the virtualized RAN ecosystem," *IEEE Commun. Standards Mag.*, vol. 5, no. 4, pp. 96–103, Dec. 2021.
- [3] Cisco, Rakuten, Altostar. (2019). *Reimagining the End-to-End Mobile Network in the 5G Era*. [Online]. Available: <https://www.cisco.com/c/dam/en/us/solutions/service-provider/mobile-internet/pdfs/reimagining-mobile-network-white-paper.pdf>
- [4] G. Garcia-Aviles et al., "Nuberu: Reliable RAN virtualization in shared platforms," in *Proc. 27th MobiCom*, 2021, pp. 749–761.
- [5] H. Reading, "5G transport: A 2021 heavy reading survey," Tech. Rep., Feb. 2022.
- [6] N. DOCOMO. (2013). *DOCOMO to Develop Next-Generation Base Stations Utilizing Advanced C-RAN Architecture for LTE-Advanced*. [Online]. Available: [https://www.ntt-docomo.com/english/info/media\\_center/pr/2013/0221\\_00.html](https://www.ntt-docomo.com/english/info/media_center/pr/2013/0221_00.html)
- [7] Ericsson. (2021). *Exploring New Centralized Ran and Fronthaul Opportunities*. [Online]. Available: <https://www.ericsson.com/en/blog/2021/5/exploring-new-centralized-ran-and-fronthaul-opportunities>
- [8] AT&T. (Feb. 2022). *Cloudifying 5G With an Elastic RAN*. [Online]. Available: <https://about.att.com/innovationblog/2022/cloudifying-5g-with-elastic-ran.html>
- [9] *Cloud Architecture and Deployment Scenarios for O-RAN Virtualized RAN (O-RAN.WG6.CADS-v04.00)*, O-RAN Alliance, Alfter, Germany, Oct. 2022.
- [10] A. Tootoonchian et al., "ResQ: Enabling SLOs in network function virtualization," in *Proc. 15th USENIX NSDI*, 2018, pp. 283–297.
- [11] A. Manousis et al., "Contention-aware performance prediction for virtualized network functions," in *Proc. ACM SIGCOMM*, 2020, pp. 270–282.
- [12] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 43–56.
- [13] P. Kumar et al., "PicNIC: Predictable virtualized NIC," in *Proc. ACM SIGCOMM*, 2019, pp. 351–366.
- [14] J. Gong et al., "Microscope: Queue-based performance diagnosis for network functions," in *Proc. ACM SIGCOMM*, 2020, pp. 390–403.
- [15] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith, "SrsLTE: An open-source platform for LTE evolution and experimentation," in *Proc. 10th ACM Int. Workshop Wireless Netw. Testbeds, Experim. Eval., Characterization*, Oct. 2016, pp. 25–32.

- [16] J. Ding et al., "Agora: Real-time massive MIMO baseband processing in software," in *Proc. 16th CoNEXT*, 2020, pp. 232–244.
- [17] X. Foukas and B. Radunovic, "Concordia: Teaching the 5G vRAN to share compute," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 580–596.
- [18] J. A. Ayala-Romero et al., "vraIn: A deep learning approach tailoring computing and radio resources in virtualized RANs," in *Proc. 25th MobiCom*, 2019, pp. 1–16.
- [19] J. A. Ayala-Romero, A. Garcia-Saavedra, M. Gramaglia, X. Costa-Pérez, A. Banchs, and J. J. Alcaraz, "VrAIn: Deep learning based orchestration for computing and radio resources in vRANs," *IEEE Trans. Mobile Comput.*, vol. 21, no. 7, pp. 2652–2670, Jul. 2022.
- [20] Y. Y. Chun, M. H. Mokhtar, A. A. A. Rahman, and A. K. Samingan, "Performance study of LTE experimental testbed using OpenAirInterface," in *Proc. 18th Int. Conf. Adv. Commun. Technol. (ICACT)*, Jan. 2016, pp. 617–622.
- [21] *Evolved Universal Terrestrial Radio Access (E-UTRA); Physical Channels and Modulation*, 3GPP, document TS 36.211, 2022. [Online]. Available: <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2425>
- [22] B. Jacob, D. Wang, and S. Ng, *Memory Systems: Cache, DRAM, Disk*. Burlington, MA, USA: Morgan Kaufmann, 2010.
- [23] *O-RAN-WG1-O-RAN Architecture Description—V04.00.00*, Open RAN Alliance, Alfter, Germany, Mar. 2021.
- [24] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005.
- [25] J. Khalid et al., "Iron: Isolating network-based CPU in container environments," in *Proc. 15th USENIX NSDI*, 2018, pp. 313–328.
- [26] *Evolved Universal Terrestrial Radio Access Network (E-UTRAN); SI General Aspects and Principles*, 3GPP, document TS 36.410, 2022. [Online]. Available: <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2443>
- [27] *Evolved Universal Terrestrial Radio Access Network (E-UTRAN); X2 Application Protocol (X2AP)*, 3GPP, document TS 36.423, 2022. [Online]. Available: <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2452>
- [28] (2022). *Operate on Secure Computing State of the Process*. [Online]. Available: <https://man7.org/linux/man-pages/man2/seccomp.2.html>
- [29] D. Skarlatos, Q. Chen, J. Chen, T. Xu, and J. Torrellas, "Draco: Architectural and operating system support for system call security," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 42–57.
- [30] Intel CAT. (Apr. 2015). *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Intel Corporation. [Online]. Available: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf>
- [31] J. Patterson, "Modern microprocessors: A 90 minute guide!" *Cortex*, vol. 15, p. A57, Jan. 2003.
- [32] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc.*, vol. 11, p. 2007, Nov. 2007. [Online]. Available: <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf>
- [33] B. Gregg, *Systems Performance: Enterprise and the Cloud*. London, U.K.: Pearson, 2014.
- [34] O-RAN Alliance. (2022). *O-RAN O2 General Aspects and Principles 2.0*. [Online]. Available: <https://orandownloadsweb.azurewebsites.net/specifications>
- [35] V. Mnih et al., "Playing Atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.
- [36] D. Raposo, A. Santoro, D. Barrett, R. Pascanu, T. Lillicrap, and P. Battaglia, "Discovering objects and their relations from entangled scene representations," 2017, *arXiv:1702.05068*.
- [37] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
- [38] J. A. Ayala-Romero, A. Garcia-Saavedra, X. Costa-Pérez, and G. Iosifidis, "Bayesian online learning for energy-aware resource orchestration in virtualized RANs," in *Proc. IEEE Conf. Comput. Commun.*, May 2021, pp. 1–10.
- [39] J. A. Ayala-Romero, A. Garcia-Saavedra, X. Costa-Pérez, and G. Iosifidis, *EdgeBOL: Automating Energy-Savings for Mobile Edge AI*. New York, NY, USA: Assoc. Comput. Machinery, 2021, pp. 397–410, doi: [10.1145/3485983.3494849](https://doi.org/10.1145/3485983.3494849).
- [40] *Evolved Universal Terrestrial Radio Access (E-UTRA); Physical Layer Procedures*, 3GPP, document TS 36.213, Jun. 2022. [Online]. Available: <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2427>
- [41] A. Jaiantilal, Y. Jiang, and S. Mishra, "Modeling CPU energy consumption for energy efficient scheduling," in *Proc. 1st Workshop Green Comput.*, Nov. 2010, pp. 10–15.
- [42] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky, "Energy of computing on multicore CPUs: Predictive models and energy conservation law," 2019, *arXiv:1907.02805*.
- [43] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1440–1448.
- [44] J. Lei Ba, J. Ryan Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.
- [45] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
- [46] G. Thimm and E. Fiesler, "High-order and multilayer perceptron initialization," *IEEE Trans. Neural Netw.*, vol. 8, no. 2, pp. 349–359, Mar. 1997.
- [47] S. Tripathi, C. Puligheddu, S. Pramanik, A. Garcia-Saavedra, and C. F. Chiasserini, "Fair and scalable orchestration of network and compute resources for virtual edge services," *IEEE Trans. Mobile Comput.*, pp. 1–17, 2023.
- [48] J. S. Panchal, S. Subramanian, and R. Cavatur, "Enabling and scaling of URLLC verticals on 5G vRAN running on COTS hardware," *IEEE Commun. Mag.*, vol. 59, no. 9, pp. 105–111, Sep. 2021.
- [49] A. Panda et al., "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX OSDI*, 2016, pp. 203–216.
- [50] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for Linux applications," in *Proc. Cloud Comput. Secur. Workshop*, Nov. 2021, pp. 139–151.
- [51] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manage.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.
- [52] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–22, Oct. 2017.
- [53] J. Martins et al., "ClickOS and the art of network function virtualization," in *Proc. 11th USENIX NSDI*, 2014, pp. 459–473.
- [54] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 1–36, Nov. 2015.
- [55] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 1–39, Mar. 2018.
- [56] A. Scolari, D. B. Bartolini, and M. D. Santambrogio, "A software cache partitioning system for hash-based caches," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 1–24, Dec. 2016.
- [57] W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely Jr., and J. Emer, "The gradient-based cache partitioning algorithm," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–21, Jan. 2012.
- [58] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with Intel's cache allocation technology," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2017, pp. 194–205.



**Josep Xavier Salvat Lozano** (Member, IEEE) received the Ph.D. degree from the Technical University of Kaiserslautern in 2022. He is currently a Senior Research Scientist with the 6G Network Group, NEC Laboratories Europe, Heidelberg. He has actively participated in several EU-founded projects, including H2020 5G-Crosshaul, H2020 5G-Transformer, and H2020 5Growth. His research interests include the application of machine learning to real-life computer communications systems, including resource allocation and energy efficiency problems. He was a reviewer of several international scientific conferences and journals, including IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE ICC, and Computer Communications journal.



**Andres Garcia-Saavedra** received the Ph.D. degree from the University Carlos III of Madrid (UC3M) in 2013. Then, he joined Trinity College Dublin (TCD), Ireland, as a Research Fellow. Since July 2015, he has been with NEC Laboratories Europe, where he is currently a Principal Research Scientist. He has published at top research venues, such as IEEE INFOCOM or ACM MobiCom, and holds several patents. His research interests include the application of fundamental mathematics to real-life wireless communication systems. He is a 5GPPP

Technology Board Member. He has served on the Program Committee and Editorial Team of several conferences and journals, such as IEEE ICC or IEEE TRANSACTIONS ON NETWORK SCIENCE AND ENGINEERING.



**Xavier Costa Perez** (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in telecommunications from the Polytechnic University of Catalonia (UPC), Barcelona, Spain. He is currently the Scientific Director of the i2Cat Research and Development Centre, the Head of the Beyond 5G Networks Research and Development at NEC Laboratories Europe, and a Research Professor at ICREA. His team contributes to product roadmap evolution as well as European Commission research and development collaborative projects. In addition, the team contributes to related standardization bodies, such as 3GPP, O-RAN, ETSI RIS, and IETF. He has published at top research venues and holds several patents. He has been a 5GPPP Technology Board Member. He received several awards for successful technology transfers and the National Award for his Ph.D. thesis. He served on the Program Committee of several conferences, including IEEE Greencom, WCNC, and INFOCOM. He serves as an Editor for IEEE TRANSACTIONS ON MOBILE COMPUTING and IEEE TRANSACTIONS ON COMMUNICATIONS journals.



**Xi Li** received the M.Sc. degree from the Technical University of Dresden in 2002 and the Ph.D. degree from the University of Bremen, Germany, in 2009. She is currently a Senior Researcher of 6G Networks Research and Development at NEC Laboratories Europe, Germany, and the Vice Chairman of the 5GPPP Architecture Working Group. Previously, she was a Senior Researcher Fellow and a Lecturer with the University of Bremen and a Solution Designer at Telefonica, Germany. She has published more than 80 journals and conference publications, and given

many invited talks in various industrial events and international conferences. She is an inventor of 18 patents, including seven granted patents, and is active in contributing to IETF CCAMP WG with two published RFCs. She is the Technical Manager of the EU H2020 5Growth Project. From 2015 to 2019, she has led technical work package in EU H2020 5G-Crosshaul and 5G-TRANSFORMER projects. Her research interests include the design of next-generation mobile and wireless networks, open and virtualized RAN, distributed edge platform solutions, and applying AI/ML for resource, service management, and automation. She received the Best Overall Award from the IETF'99 Hackathon in 2017.