IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 July 2024, accepted 29 July 2024, date of publication 12 August 2024, date of current version 26 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3441725

RESEARCH ARTICLE

Control Plane Performance Benchmarking and Feature Analysis of Popular Open-Source 5G Core Networks: OpenAirInterface, Open5GS, and free5GC

TARIRO MUKUTE[1], LUSANI MAMUSHIANE[1,2], ALBERT A. LYSKO[1,2], (Senior Member, IEEE), ELENA-RAMONA MODROIU[3], THOMAS MAGEDANZ[3], (Senior Member, IEEE), AND JOYCE MWANGAMA[1], (Member, IEEE)

[1]Department of Electrical Engineering, University of Cape Town, Cape Town 7700, South Africa
[2]Council for Scientific and Industrial Research, Pretoria 0184, South Africa
[3]Department of Next Generation Networks, Technische Universität Berlin, 10587 Berlin, Germany

Corresponding author: Tariro Mukute (mkttar001@myuct.ac.za)

ABSTRACT This study examines the maturity and state of open-source 5G Core (5GC) networks, with a focus on their support of 5G procedures and Network Function (NF) operations. The research emphasises the importance of optimising the virtualised resource usage of 5GC control plane functions. Given the large set of NF operations and procedures defined for 5GC networks, the study presents a framework that correlates NF operations with 5G procedures, thus facilitating the assessment of 5GC implementations. Furthermore, the study introduces macro-benchmarking and micro-benchmarking approaches to assess the performance of 5GC control plane functions. Our 5G traffic generator is used to generate User Equipment (UE) registration traffic for benchmarking. Macro-benchmarking measures the time taken for a specific number of UEs to complete UE procedures. In contrast, micro-benchmarking analyses the system calls made by the 5GC, presenting their latency and frequency and the processes responsible for generating them. Our findings revealed a direct correlation between macro-level performance and micro-benchmarking results. This suggests that optimising micro-level implementations is crucial for enhancing overall system performance. Moreover, examining micro-benchmarking data can provide valuable insights into how the system's performance will scale under varying workloads. We use the correlation of benchmarking results to identify performance improvement points in each 5GC and to provide recommendations for software architectural changes that can optimise the usage of virtualised resources. The study includes artefacts and source code to enable replication of the work and results.

INDEX TERMS 5G, 5G core networks, BPF compiler collection tools (BCC Tools), free5GC, Linux Kernel, load testing, network function virtualisation, network traffic generator, OAI, OpenAirInterface, Open5GS, open source software performance benchmarking, system calls.

I. INTRODUCTION

Fifth-generation wireless (5G) is the latest iteration of mobile communication technology, designed to increase the speed and responsiveness of mobile networks in an unprecedented way. In particular, 5G promises to enable sophisticated use cases, such as connected vehicles, remote surgery, remote control of production lines, and retail and medical drone deliveries. To meet 5G performance objectives, the Third Generation Partnership Project (3GPP) has defined new network architectures such as the next-generation core network, known as the 5G core (5GC) [1], and the new Radio

The associate editor coordinating the review of this manuscript and approving it for publication was Tarcisio Ferreira Maciel.

Access Technology (RAT), known as the 5G new radio (5G NR) [2]. For the 5th Generation Core Network (5GC), the 3GPP introduced Control and User Plane Separation (CUPS). In CUPS for 5GC, the control and data planes are separated, ensuring efficient and scalable network operations.

The 5GC control plane is considered the heart of the 5G mobile network and handles essential Network Functions (NFs) such as connectivity, mobility management, session management, authentication, authorisation, data aggregation, and many others. Compared to previous generations of core networks (such as 4G/LTE or 3G), 5GC control plane has been purposefully designed to support a significantly larger number of procedures and operations. For example, the 5GC control plane encompasses procedures specifically designed to facilitate network slicing operations, including instantiation, monitoring, and decommissioning of network slices. Additionally, it incorporates NF discovery and management operations that enable functionalities such as registering and removing NF instances. The time to complete these control plane operations directly impacts the delay experienced by end-user applications [3], which makes the performance of the control plane pivotal to 5GC performance.

To deliver both agility and flexibility, 5GC networks adopt a cloud-native approach and are designed to be infrastructure-agnostic. This architectural choice enables rapid 5G innovation and is necessary to unlock new capabilities such as on-demand network slice customisation. Furthermore, the cloud-native architecture means that the core network can be deployed on any cloud — private, public or hybrid cloud, centralised or edge cloud — depending on service requirements. Although being cloud-native offers numerous advantages, it can also lead to performance degradation due to virtualisation of computing resources [4], [5]. This is a significant concern in 5GC networks, which must be addressed to unlock the promises of 5G fully. One way to address this problem is to ensure that 5GC Virtual Network Function (VNF)s are designed with performance costs in mind. To achieve this, the implementation of 5GC must include making software architectural decisions that optimise the use of virtualised resources.

Although extensive research has investigated the performance of the 5G Core Data Plane (User Plane Function (UPF)) in various studies, e.g., [6], [7], and [8], the analysis of its Control Plane (CP) performance remains fragmented. The literature search identified only three published studies closely related to the 5G Control Plane performance evaluation. Some efforts address TLS optimisation for CP network function communication [9] or performance improvements through shared memory [10]. In another study, [3], the focus is on scenarios and challenges that impact CP performance, such as UE-Core state inconsistency, slow state updates, and frequent control handovers. To bridge this gap, our work presents the first study of its kind: an in-depth performance benchmarking and comparative analysis of different open-source 5GC implementations focused solely on the control plane.

A plethora of open-source and 3GPP-compliant 5GC standalone (SA) implementations have emerged since the introduction of 5G. Some notable implementations include OpenAirInterface (OAI) [11], free5GC [12], Open5GS [13], and Open5GCore [14]. These implementations can run on different types of Network Function Virtualisation infrastructures (NFVis) such as containers, pods, or virtual machines. The aim of this study is to assess and compare the runtime performance and feature support of open-source 5G core networks. The focus of our performance evaluation is specifically on the control plane network functions. In terms of feature support assessment, various factors will be considered, including the adoption coverage of each 5GC, the maturity level indicated by the number of network operations it supports, the minimum hardware requirements necessary to run the 5GC, the licensing model employed, and other relevant factors.

This study evaluates three different 5GC implementations, i.e. OpenAirInterface (OAI), free5GC and Open5GS. We assess the performance of registration scenarios for three open-source 5G core networks using two benchmarking approaches, namely, macro-benchmarking and micro-benchmarking introduced in [15], [16], and [17]. The macro-benchmarking provides end-to-end performance of 5GC based on metrics such as the time taken for a given number of User equipment (UE)s to register and the Round-Trip Time (RTT) for the Stream Control Transmission Protocol (SCTP) association during registration. On the other hand, micro-benchmarking evaluates the *latency* and *frequency* of the underlying operations of the 5GC control plane functions by analysing system calls and their patterns.

To assess performance, each 5GC network (which constitutes the system under test) is traffic-loaded with UE registrations generated by our 5GC traffic generator [18]. To gain insight into the software architectural design of each 5GC, we complemented the micro-benchmarking with system profiling.[1] This helped us identify areas for performance improvement and make recommendations accordingly. When applicable, we also included tracing, which emphasises the temporal aspect of performance variations and reveals the specific points in the code where optimal performance can be attained [19]. To conduct the micro-benchmarking, we used Linux high observability[2] tools, which enabled us to understand the underlying software architecture of 5G core implementations without having to go through the code base.

In addition to performance, another critical factor to consider when evaluating the open-source 5GC is the level of support for 3GPP releases. These releases specify the use cases and features that the 5GC can support, along with the

[1] Profiling presents summary statistics of performance metrics [19].

[2] Observability is the ability to monitor, measure, and understand the state of a system or application by examining its outputs, logs, and performance metrics.

corresponding procedures required to implement these features. The procedures consist of a series of ordered operations that must be performed by control plane network functions to fulfil them. However, there are numerous procedures and their respective operations, making the assessment process a challenging task. In this study, we introduce a framework for assessing the extent of 5GC support and provide a summary of our assessment.

II. CONTRIBUTIONS

Our study aims to contribute:

1) A feature-based evaluation of three open-source 5GC implementations, namely Open5GS, free5GC, and OpenAirInterface (OAI) — by comparing the features of each 5GC, practitioners in the 5G domain can make informed decisions on which 5GC implementation will best meet their specific requirements. Furthermore, a feature-based evaluation can help identify the gaps in feature support between different 5GC implementations, which can aid future development and improvement efforts.
2) A framework that evaluates the software architecture of three of the five open-source core networks. This framework considers how software architectural decisions affect the performance of 5GC networks on virtualised platforms. Performance evaluation and benchmarking are crucial to identify performance bottlenecks to determine if the network is capable of meeting the performance requirements of specific use cases, and to determine the maximum traffic load that the network can handle.
3) Based on our evaluations, we provide recommendations for architectural changes in the software that can be made to minimise performance degradation due to the virtualisation of 5GC networks — these recommendations are aimed at fostering continuous improvement efforts by the research community working on 5GC networks. It is important to note that these recommendations are not limited to open-source projects, but can also be implemented by 5GC vendors and network operators to optimise their 5G core network solutions.

In addition to the main contributions, this study also includes comprehensive tutorials and related source codes to reproduce the benchmark experiments [20]. These additional contributions can be summarised as follows:

1) *Ansible* roles and playbooks for setting up and replicating the benchmarking environment;
2) A tutorial on how to replicate the experiment along with the data collected from the experiment;
3) Contribution to the BPF Compiler Collection (*BCC*) tools code base to enable easier visualisation of the results for comparison;
4) *Jupyter Notebook* for analysing and visualising the results from *BCC* tools

III. 5G CORE NETWORK ARCHITECTURE

The architecture of 5G is designed to be cloud-native, with the aim of achieving greater agility and programmability. The Next-Generation Core (NG Core) network, also known as the 5GC, was specified in 3GPP Release 15 [21], and constitutes network functions, such as Access and Mobility Management Function (AMF), Session Management Function (SMF), UPF, and Policy Control Function (PCF), each designated to perform specific functionalities as outlined in 3GPP Technical Specification (TS) 23.501 [22]. The 3GPP Release 15 release presented two NG Core architectures: the point-to-point architecture and the service-based architecture. The Service-Based Architecture (SBA), which is closer to the cloud-native concept, was chosen for its flexibility in system updates and a short time to market for new services. In the service-based architecture, a services model is used instead of predefined interfaces between elements, enabling network functions to query an NF Repository Function (NRF) to discover and communicate with each other. For the rest of this study, when we state 5GC, we will be referring to a 5GC that has been designed using a service-based architecture (SBA).

The 5GC introduces two main changes from 3GPP Release 15 [21]: CUPS as in software-defined network (SDN) and decomposition of the system into microservices as in cloud-native design. The microservices architecture, also referred to as SBA, allows NFs to interact using uniform interface connections known as Service-Based Interface (SBI), allowing NFs to be extended to any service-permitted consumer. Therefore, the SBA offers the reusability and modularity of NFs [23]. The 5GC is effectively a collection of NFs separated into control and user planes that interact through defined interfaces. The 3GPP TS 23.501 standard for the system architecture for the 5G System [24] in Sections VI and VII gives an inventory of the NFs for the 5GC as defined in Release 17.

Each NF comprises smaller unit functions known as NF services, which interact through SBI interfaces using a producer-consumer model. The NFs make a set of transactions to achieve a specific goal or result, such as the UE registration. These transactions, or series of operations, are referred to as Procedures. Procedures essentially entail the 5GC features and functionalities and how they can be delivered. In order to know the functionality delivered by a core network, we can look at the procedures it supports. The following are some of the most important 3GPP technical specifications that cover the procedures for 5G systems:

- **3GPP TS 23 502** [25]: This standard specifies the overall architecture and procedures for the 5G system, including the core network, the radio access network, and user equipment. It also defines the interfaces between different network functions and the protocols used for communication.
- **3GPP TS 23.527** [26]: Specifically, Section 4.6 of this standard describes the restoration procedures for the 5G core network.

- **3GPP TS 32.290** [27]: This standard specifies procedures for charging and policy control in the 5G SBI.
- **3GPP TS 33.501** [28]: This standard outlines the security architecture (features, requirements, mechanisms, etc.) and procedures for 5G networks.

IV. FEATURE-BASED EVALUATION OF 5GC NETWORKS

Evaluating or comparing the features and procedures supported by open-source 5G core networks can be a tedious task due to the multitude of procedures involved. For example, in Release 17 alone (as described in 3GPP TS 23 502 [25]), there are about 126 procedures and more than 596 operations. However, it is crucial to perform such evaluations for performance improvement and 5GC conformance assessment purposes. Furthermore, evaluation plays a vital role in understanding the appropriate tools to utilise and the extent of work required when implementing a research idea, as well as selecting the most suitable core network for specific scenarios.

To simplify the evaluation process and aid in decision-making on the adoption of a 5G core network, we conduct a feature-based comparison. Specifically, we compare three open-source 5G core implementations: Open5GS, free5GC, and OpenAirInterface (OAI). The scope of our evaluation is on the following broad categories of features:

- Community support and adoption coverage of each open-source 5G core network by examining and comparing project statistics, including fork count, commits, number of contributors, watchers, and issues as described in Table 1.
- Software and hardware requirements and capabilities of the open-source core network implementation by evaluating and comparing the minimum setup requirements, as well as capabilities such as support for NB-IoT, and others, as indicated in Table 2.
- Supported procedures of a given 5G core network and comparing them with the procedures supported by each implementation of open-source 5G core networks, as presented in Table 3.

To evaluate the procedures supported by a given 5G core network and to compare the features and procedures supported by open-source 5G core networks, we implemented a project [29]. The project implemented the assessment using the following steps, which can be easily reproduced for other 5G core networks:

1) We generated a checklist of all the NF operations defined by Release 17. We generated this checklist from application program interface (API) documentation artefacts by 3GPP.[3]
2) For each 5GC, we create an inventory file of the NF operations by going through the source code and updating the checklist from step (1), ticking the operations supported. We use the resultant inventory file as a variables file in the next steps. Since there are many NF operations, we make the resulting inventory list of the core networks available on the webpages[4] [29]. Table 3 gives a summary of the number of operations supported, only for the operations for the NF implemented by the core networks.
3) We create procedures sequence flow using *plantuml*, an open-source diagrams-as-code tool, which allows us to make colours dynamically represent a connection (to indicate whether an operation is supported or not). In addition to making the connection colours dynamic, *plantuml* allows us to merge the related procedures sequence diagrams into a single diagram, making it easier to evaluate connected procedure flows.
4) For each of the core networks, we generate a set of procedures (using the inventory file from step (2)) as variables that indicate whether an operation in a procedure is supported or not. Since the results from this are pages long, we created a webpage to side-by-side depict the difference for each procedure between the core networks [29]. We provide a sample of the registration procedure diagrams in the Appendix section in figs. 14 to 16.

V. NETWORK FUNCTION VIRTUALISATION

Virtual Network Functions (VNFs) are network functions that run on virtualised platforms. Virtualisation allows NFs to run on commercial off-the-shelf infrastructure. There are different virtualisation approaches, the most common being hypervisor virtualisation and OS-level virtualisation [30], [31]. Regardless of the manner of virtualisation employed, in most cases, a VNF is a piece of software that runs on top of the Linux kernel, and this degrades its performance compared to the hardware NF [4].

To gain a better understanding of how virtualisation affects performance and how the software architecture of VNFs affects the performance of the 5G core networks, it is necessary to dive deeper into the underlying operations of the Linux kernel. A standard Linux system is based on the monolithic kernel architecture, which divides the system into two main components: kernel-space and user-space, which operate at different privilege levels. The kernel-space is responsible for managing the hardware and software components of the system and accounts for independent operations of programs. The user-space contains and executes user-defined programs and data. These two spaces are kept separate from each other, and communication between them is only possible through secure system calls. This concept allows the Linux kernel to abstract away the complexities of interacting directly with the hardware, which requires specialised communication [32].

The system calls provide abstract APIs for various functionalities, including process management, memory management, file systems, device drivers, and networking. The kernel manages and executes these system calls by

[3]https://forge.3gpp.org/rep/all/5G_APIs.git

[4]https://tariromukute.github.io/5gc-features

TABLE 1. Community support and adoption coverage.

5G Core	Age	Forks	Commits	Watchers	Contributors	Last Activity Update	Issues
free5GC	3y	465	96	86	27	<1 month	88
Open5GS	12y	411	3 694	96	54	<1 month	630
OAI	6y		10 023		97	<1 month	567

TABLE 2. Software, hardware and capability assessments of open-source 5G core networks.

Feature	free5GC	Open5GS	OAI 5GC
3GPP Release	15	17	16
CPU Architecture	ARM & x86	ARM & x86	ARM & x86
Minimum Hardware Requirements	RAM: 4GB HDD: 160GB 1 CPU	RAM: 4GB HDD: 10 GB 1 CPU	RAM: 4GB HDD: 40GB 2 CPUs
Operating System	Linux & Windows	Linux (Debian & OpenSUSE)	Linux (Red Hat & Ubuntu)
Programming Language	Go	C	C++ & C
Licensing Model	Apache 2.0	GNU AGPL-3.0	OAI Public License-1.1
Version Number	v3.3.0	v2.6.4	v1.5.1 (2023.w34)
NB-IoT Support	No	No	Partly
Database	MongoDB	MongoDB	MySQL
Docker Support	Yes	Yes	Yes

interfacing with the software or hardware support modules, as described and illustrated in [33]. Of importance to the scope of this study is process management.

A. PROCESS MANAGEMENT

In Linux, a process refers to an instance of a computer program that is currently running and comprises various components, including an identifier, state, priority, program counter, memory pointer, context data, and Input and Output (I/O) request [34]. Managing these processes involves scheduling, switching between states, and allocating resources [35]. Central Processing Unit (CPU) scheduling is the responsibility of the CPU scheduler, which prioritises processes based on their importance or urgency using a specific scheduling policy. A process is executed on the CPU as a set of tasks, also referred to as threads. Threads may exit the CPU (a) involuntarily if they exceed their allocated CPU time or are preempted by higher-priority threads, or (b) voluntarily if they block an I/O operation, a lock, or a sleep. When threads block, they remain suspended until they receive a wake-up event [35]. The CPU scheduler's management of threads is illustrated in Figure 1a, showing the various states in which threads can be. Blocking is a problem for programs that should operate concurrently, since blocked processes are suspended. Concurrent programs need to use non-blocking alternatives to prevent performance issues from blocking processes.

Processes can be cloned to share resources and function as threads within a single process. Linux does not differentiate between threads and processes [36]. Cloned threads that share resources require locking mechanisms to prevent race conditions from concurrent resource access. Locks can be user-level locks or kernel-level locks, with different types available in the Linux kernel, such as spin locks, *mutex* (mutual exclusion) locks, and reader-writer locks. However, the use of locks may affect performance due to thread blocking [35].

VI. PERFORMANCE-BASED EVALUATION OF 5GC NETWORKS

The centre of operations between the application (VNF) and the kernel, along with its underlying resources, is occupied by system calls, as shown in Figure 1b. By analysing these system calls, it is possible to gain insight into the unique

TABLE 3. Number of operations supported by 5GC implementation. Blue font highlights instances where all operations have been supported, and bold font highlights the largest number of operations per line.

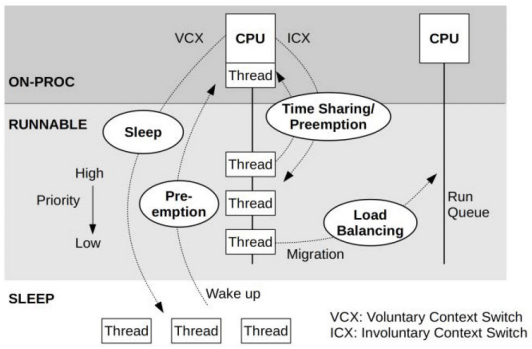
3GPP Release 17 5GC NFs Inventory		Number of operations supported		
Network Function	**Network Function service**	Open5GS	free5GC	OpenAirInterface (OAI)
AMF	Communication	1/16	**10/16**	3/16
	EventExposure	0/3	3/3	2/3
	MT	0/3	**1/3**	0/3
	Location	0/3	**1/3**	0/3
SMF	PDUSession	3/10	3/10	3/10
	EventExposure	0/4	0/4	**1/4**
	NIDD	0/1	0/1	0/1
PCF	AMPolicyControl	2/4	4/4	0/4
	SMPolicyControl	2/4	4/4	4/4
	PolicyAuthorization	3/5	5/5	0/5
	BDTPolicyControl	0/3	3/3	0/3
	UEPolicyControl	0/4	0/4	0/4
	EventExposure	0/4	0/4	0/4
	AMPolicyAuthorization	0/6	0/6	0/6
	MBSPolicyControl	0/4	0/4	0/4
	MBSPolicyAuthorization	0/4	0/4	0/4
UDM	UECM	2/31	**7/31**	2/31
	SDM	6/36	**15/36**	4/36
	UEAuthentication	**3/7**	2/7	**3/7**
	EventExposure	0/3	3/3	3/3
	ParameterProvision	0/8	**1/8**	0/8
	NIDDAuthorisation	0/1	0/1	0/1
	ServiceSpecificAuthorisation	0/2	0/2	0/2
	ReportSMDelivery	0/1	0/1	0/1
NRF	NFManagement	7/8	7/8	8/8
	NFDiscovery	1/6	1/6	**3/6**
	AccessToken	1/1	1/1	1/1
AUSF	UEauthentication	**4/10**	3/10	2/10
	SoRProtection	0/1	0/1	0/1
NEF	EventExposure	0/4	0/4	4/4
	PFDManagement	0/6	0/6	0/6
	SMContext	0/4	0/4	0/4
	Authentication	0/1	0/1	0/1
	EASDeployment	0/3	0/3	0/3
UDR	DM	0/5	0/5	0/5
	GroupIDmap	0/5	0/5	0/5
NSSF	NSSelection	1/1	1/1	1/1
	NSSAIAvailability	0/7	**5/7**	**1/7**
BSF	Management	**4/15**	0/15	0/15
Total number of operations Supported		**40/244**	**80/244**	**45/244**

characteristics of VNF workloads. As discussed in the Results and Analysis section, examining usage patterns and data related to system calls can also provide valuable information on the software architecture of VNFs or systems. In this study, we use this information to evaluate the performance of open-source 5GC implementations and make recommendations for software architecture improvements applicable to each core network.

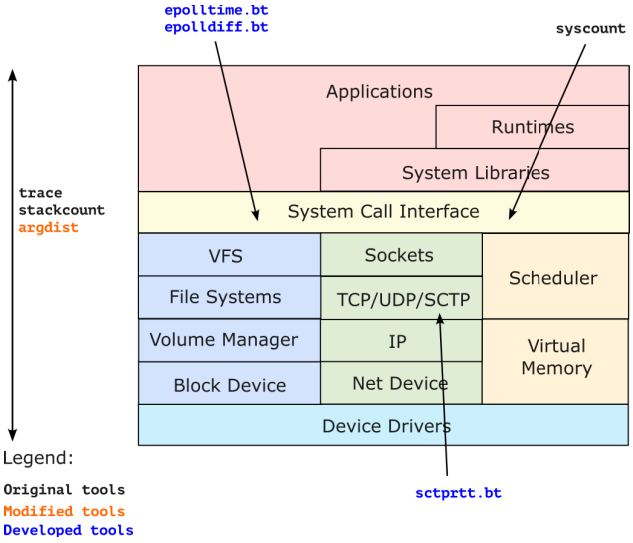
A. RELATED WORKS

Benchmarking refers to the process of conducting experiments and measurements on a System Under Test (SUT) in order to collect data that help to understand its behaviour [37]. However, accurately benchmarking a system can be challenging, especially as systems evolve from batch-style execution to multiple parallel processes managed and orchestrated by a general-purpose system such as the Linux kernel. This evolution has made benchmarking challenging, necessitating alternative approaches to benchmarking [17], [37], [38]. Most existing benchmarking approaches attempt to represent the overall performance of the system as a scalar quantity [16]. However, this approach is suitable for simpler systems but may not be adequate for accomplishing performance evaluations of more complex systems.

Several studies have explored alternative approaches to benchmarking. For example, the study [15] suggests



(a) CPU scheduler [35]



(b) BPF Performance Tools used in this study and their visibility

FIGURE 1. Linux CPU scheduler and BPF Performance tools.

decomposing benchmarking into (i) macro-benchmarking, which measures end-to-end performance on a specific process or workload, and (ii) micro-benchmarking, which employs a more granular approach by individually measuring very small pieces of the Operating System (OS) kernel's abstraction layers. The OS kernel's abstraction follows a layered structure consisting of two main layers. Firstly, the application layer interfaces directly with the OS. Secondly, the OS layer interfaces with the underlying hardware, providing a bridge between the application layer and the hardware components of the system. This decomposition process allows for a thorough examination of system performance at different abstractions. It also helps OS developers identify the abstractions that are critical for the application's performance. Additionally, application developers can utilise this benchmarking approach to optimise their usage of OS abstractions, such as system calls, in relation to hardware. Furthermore, by examining the abstractions, hardware manufacturers can explore various methods of hardware tuning to enhance overall hardware performance.

The authors in [16] introduce three benchmarking methodologies, including the vector-based approach, which is similar to the decomposition approach described in [15]. The vector-based approach involves characterising an underlying system, such as an operating system or Java Virtual Machine (JVM), by utilising a set of micro-benchmarks. These micro-benchmarks describe the behaviour of fundamental primitives of the system. The fundamental principle underlying the vector-based approach is based on the observation that different primitive operations in a typical computer system operation have varying completion times [16].

Another study [17] emphasises the importance of micro-benchmarks and focusses on measuring safety-critical latency on multi-core systems, by examining sub-latencies in the communication between processes and hardware. The authors of the study [37] acknowledge the connection between overall performance and underlying performance metrics, such as memory usage versus decoder bytes. However, it does not take into account the impact of operations that affect the request, such as the latency of the OS system call involved in reading bytes from memory.

In our study, we incorporate both macro-benchmarking and micro-benchmarking approaches similar to those described in [15], [16], and [17]. To evaluate the performance of 5G core networks, we employ macro-benchmarking, which provides end-to-end performance metrics such as the time taken for a given number of UEs to register. To achieve this, we load-test the core network using our Traffic Generator and analyse the macro-benchmark metrics.

To delve deeper into the latency and frequency of underlying system operations, we leverage micro-benchmarking techniques. This approach involves analysing system calls, representing low-level communications between a process and the kernel. Numerous observability tools are available for this analysis on Linux systems. The tools considered by us are listed and inter-compared in Table 4, based on the findings of various studies [39], [40], [41]. Taking into account the combined advantages summarised in Table 4, we leverage *eBPF* with the *BCC* and *bpftrace* tools in this study. Figure 1b further illustrates the specific tools used and their respective focal areas.

In addition to evaluating the performance of 5G core networks using the micro-benchmarking approach, we leverage micro-benchmarking to identify areas for performance improvement, which is one of the advantages of micro-benchmarking mentioned in [15]. We start by obtaining resource usage patterns, by examining metrics such as system call latency, frequency, and provided and returned arguments. These patterns offer insight into the software's architectural designs, allowing us to identify potential areas

TABLE 4. Comparision of Linux observability tools.

Tool	Metrics Coverage	Profiling Capabilities	Performance Overhead	Ease of Use	Learning Curve	Extensibility
BCC	High	High	Low	Moderate	Steep	High
bpftrace	High	High	Low	Moderate	Moderate	High
perf	High	High	Low	Moderate	Moderate	Moderate
ftrace	Moderate	High	Low	Moderate	Moderate	Low
SystemTap	High	High	Moderate	Moderate	Steep	Moderate
LTTng	High	High	Low	Moderate	Moderate	Moderate
sysdig	Moderate	Moderate	Moderate	Easy	Moderate	Moderate

of performance improvement. In the cases where we identify such performance improvement opportunities, we then use tracing techniques to pinpoint in the application code where these improvements can likely be applied.

B. EXPERIMENTAL SETUP AND METHODOLOGY

The objective of this study is to evaluate the performance of open-source 5G core networks by generating a high volume of control traffic, which mimics the traffic generated by UEs in real deployment scenarios, and transmitting this traffic to the 5G Core network.

In our experiment (see Figure 2), we designed and developed a Traffic Generator that emulates the behaviour of UEs and acts as a client, while the 5G core network acts as the server. The core network constitutes the SUT. To collect metrics at different vectors, the server employs *BCC* tools to monitor the resource utilisation and usage patterns of the core network. Subsequently, we analyse the performance of the 5G core network based on the usage of these resources and associated patterns. We chose to use *BCC* because it provides a more granular and dynamic view of the kernel processes. We developed a traffic generator because other traffic generators like *UERANSIM* [42] and *my5G-RANTester* [43] do not currently support significant load testing. For example, *UERANSIM* throws segmentation faults when generating traffic from 200 UEs, and *my5G-RANTester* does not support simultaneous traffic generation sessions. In addition, both *UERANSIM* and *my5G-RANTester* do not provide any performance metrics, for example, the number of UEs completing a registration procedure in a given period. We do this for all 5G core networks.

The test environment is set up as follows:

- The core networks and Traffic Generator are deployed within Virtual Machine (VM)s on an OpenStack infrastructure. The decision to use VMs was driven by the limitation of hardware availability, which prevented a dedicated hardware deployment of the 5GC and Traffic Generator. The resources allocated to the VMs running the 5GC had been selected to meet the minimum recommended specifications for the open-source 5GC being evaluated. These specifications are outlined in Table 2).
- To set up OpenStack, we used an i7 CPU@2.5GHz PC with 16 GB RAM, 8 CPUs cores, and a 200 GB HDD. We installed the *stable/zed* version of OpenStack, which includes services for networking (Neutron), computing (Nova), storage (Cinder), and dashboard (Horizon).
- Both VMs run on Ubuntu 20.04 with a kernel version of 5.4. We opted for a server Linux installation to maximise system performance, since there is no desktop environment to manage, and all compute resources are dedicated to server tasks. This is important for test performance. To minimise the degradation of system performance caused by unrelated user processes, we only installed the necessary software for the core network under evaluation, including *BCC* tools.
- The communication between the Traffic Generator VM and the Core Network VM is established through a virtual private network (VPN), which may potentially impact test performance.

To evaluate the performance of each of the three 5G core networks, we employed the steps outlined below:

1) Installation and configuration of the open source 5G networks on the server.
2) Configuration of the traffic generator tool with the respective core network details. This involved setting up subscriber information in the core network databases, configuring networking settings, and other relevant configurations.
3) Generation of control-layer traffic by emulating multiple UEs on the client side, performing the registration procedures for the UEs in a burst.
4) Collection of the macro- and micro-benchmarking data on the system metrics and usage of resources and usage patterns through *BCC* tools.
5) Analysis of the data collected to determine the performance of each open-source 5G network and gain insight into its software architectural design.

To ensure that the experiment can be easily replicated, we utilised *Ansible*, a widely used infrastructure configuration tool. With *Ansible*, we automated the setup process for the servers, installation and configuration of the Traffic Generator, installation of the core networks and *BCC* tools, and establishment of the connection between the client and the servers. We have made the necessary artefacts available on GitHub, including *Ansible* roles and playbooks for installing and configuring the test environment [20], as well as the Traffic Generator for simulating registration scenarios for

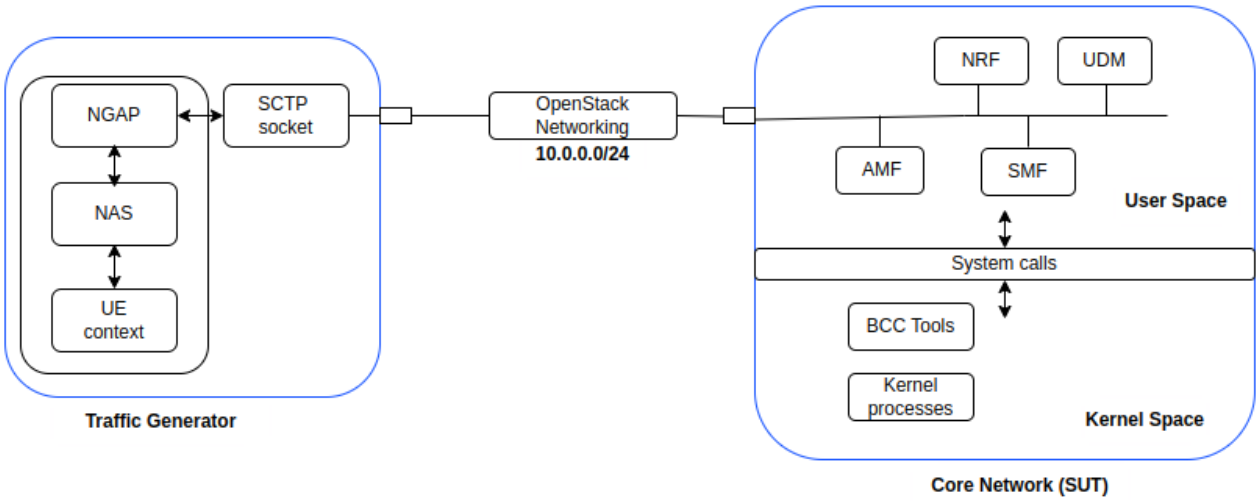


FIGURE 2. Experimental Setup.

UEs [18]. In addition, we have provided a tutorial that guides users on how to replicate the set up.

VII. RESULTS AND ANALYSIS

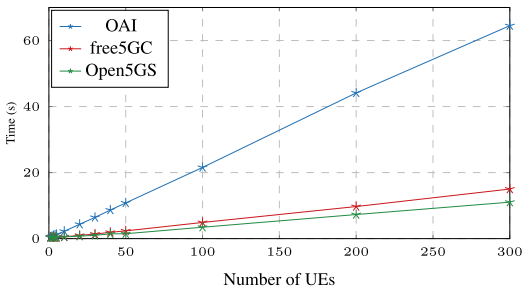
In this section, we present the findings and analysis derived from our macro- and micro-benchmarking experiments.

Benchmarking results are meant to access the performance improvement points of the 5GCs individually. Since the 5GCs have different software architectural designs, they provide non-uniform results in the micro-benchmarking section. For example, while Open5GS and OAI make use of the *select* system call, free5GC does not; it relies on the other I/O multiplexing system calls, *epoll_wait* and *poll*. In the micro-benchmarking section, we give the reader and the 5GC developers recommendations on the performance improvement points for each system call. We leave the correlation of the recommendations to the individual 5GCs to the reader and the 5GC developers.

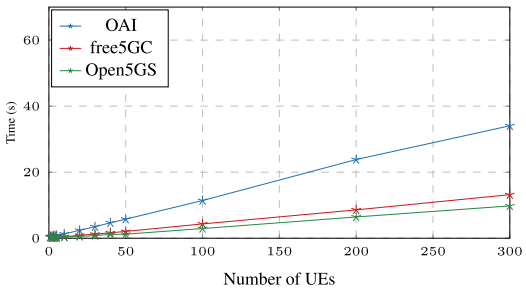
We faced challenges in generating a high load for free5GC; therefore, throughout this section, for all the core networks, we show the results up to the maximum UEs free5gc could handle - 300 UEs. Initially, we faced a similar challenge (core crashing when faced with high traffic loads) with OAI. We found the source of the problem to be the logs generated by the OAI 5GC's Non-Access Stratum (NAS) implementation, which could not be turned off. After reporting the issue, the OAI team resolved it on the latest docker images with tags *develop* and on the OAI GitLab branch tags *2023.w12*.

A. MACRO-BENCHMARKING METRICS

This section presents the high-level performance of the three 5GCs under evaluation. We present the time taken to complete the registration and de-registration procedures for a given number of UEs in Figure 3a and the average time it takes to complete registration and de-registration of a single UE in Figure 3b. In both cases, we show the results as the UE load increases. Open5GS leads in performance, with free5GC coming next and OAI trailing behind. The subsequent section on micro-benchmarking highlights specific implementation decisions that explain these outcomes.



(a) Total time for all UEs



(b) Average time per UE

FIGURE 3. Time taken to complete Registration and De-registration procedures.

B. MICRO-BENCHMARKING METRICS AND PERFORMANCE IMPROVEMENT POINTS

This section presents the results and analysis of the software architectural approaches used in each 5G core network.

The evaluation process begins with an analysis of frequently invoked system calls. Based on the categories of the system calls, we derive the most relevant virtual resources. Subsequent to this, we examine the most active processes

in the system, and for each frequently used system call, we analyse the most active processes. This approach provides valuable insights into the processes that consume the most virtual resources. During the evaluation of system calls, we consider both the latency and the frequency of calls. Here, latency refers to the time it takes the kernel to execute a given request, while frequency refers to how often a particular system call is invoked. We provide the summarised results for the system calls, for more details on how the system call usage and frequency vary for each process, we refer the reader see Figures published on our webpage[5] [20].

Furthermore, we correlate the usage pattern of each system call with the usage statistics of virtualised resources. These correlations allow us to gain insight into the architectural design of each 5G core network. Based on this analysis, we provide recommendations for performance improvement.

1) SYSCALLS ACROSS THE SYSTEM

Analysing system calls (syscalls) across the system helps in categorising the workload of the system. This information is valuable in identifying the hardware resources that require optimisation, such as installing an accelerated Network Card Interface (NIC) or a cryptographic accelerator. System calls can be roughly grouped into five major categories:

- **Process control**: These system calls are responsible for tasks such as process creation, termination, execution, loading, and so forth. They also handle memory allocation and deallocation for processes.
- **File management**: These system calls handle operations related to file manipulation, such as creating, opening, closing, deleting, reading, writing, and renaming files and directories. They also manage file attributes and permissions.
- **Device management**: These system calls are primarily responsible for requesting and releasing access to devices, such as disks, printers, and scanners. They also control device functions, such as reading and writing data from or to devices.
- **Information maintenance**: These system calls provide information on various system resources, such as time, date, user identity, and process status. They also allow for the modification of certain system parameters, such as priority levels.
- **Communication**: These system calls enable communication between processes using different methods, such as message passing or shared memory. They also support network communication through the use of sockets or pipes.

Based on the findings depicted in Figures 4, it can be confirmed that the core networks make process control and communication system calls. This information can help with hardware selection. Importantly, the results reveal that 5G core networks incur a considerable latency through *futex* system calls. Futex, short for ''fast user-space *mutex*,'' is a system call interface in Linux that provides a synchronisation mechanism primarily used for inter-process and thread synchronisation. It is an efficient alternative to traditional *mutex* implementations when the contention is expected to be low. A potential reason for the considerable latency is that, on a virtual machine, the high number of *futex* system calls may be due to the high contention on shared-memory resources, leading to many threads waiting on *futex*es. Another possible reason for high *futex* system calls is lock contention. A detailed analysis of the system calls will be presented in the following sections, providing individual discussions and insights.

[5]https://tariromukute.github.io/control_plane_performance_analysis/

2) PROCESSES MAKING SYSCALLS

The information about the processes that make system calls provides valuable insights into the most active processes during the registration procedure. By observing the changes in latency and frequency of the system calls made by a process as the number of UEs increases, we can identify processes that have a high probability of becoming bottlenecks. The information can be used to make several mitigation decisions, such as allocating more resources or dedicated resources to a given process or Network Function (NF), optimising the usage by the NF or process, and examining the configuration of the process, among other things.

Figure 5 compares the system call frequency of the top six active processes for each core network as the number of UEs increases, revealing that the frequency increases for all networks, but at different rates. The figures also review the processes that are most active during registration and de-registration procedures, with their rate of increasing eluding to the processes that can potentially become a bottleneck for the system.

Furthermore, from the results in Figure 4, we can see that Open5GS has the least number of system calls and where it is the most performant. Being the most performant implementation, we can use its system call patterns as a reference guide. We will also compare its pattern against best practices to further improve its performance.

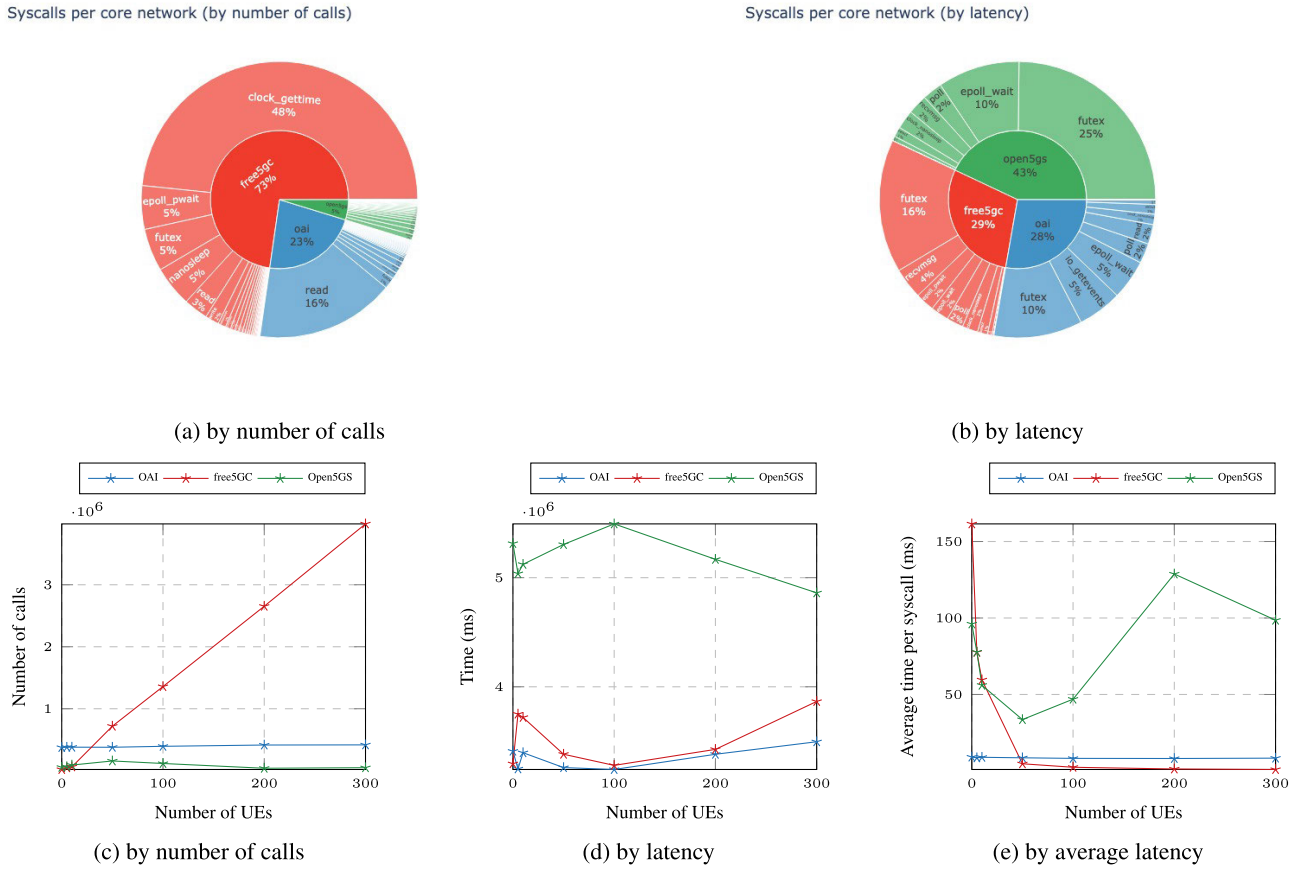
In the next section, we examine the most active system calls on the systems, and analyse how the processes make use of the system calls as the number of UEs increases. This analysis will help us gain valuable insights into the specific actions performed by a process, and enable us to identify opportunities for optimising these processes. We will also consider how each core network makes use of each of the system calls. We analyse the pattern to identify improvement points, i.e., the system calls that should not be made use of, the system calls that should be used instead and the change in configurations needed, among other things.

3) EPOLL/POLL/SELECT

The system calls *epoll*/*poll*/*select* implement I/O multiplexing, which enables the simultaneous monitoring of multiple input and output sources in a single operation. The main advantage of multiplexing I/O operations is that it avoids

TABLE 5. Socket states and blocking/non-blocking calls.

Call type	Socket state blocking	Non-blocking
Types of *read*() calls	Input is available Immediate return	Immediate return No input is available Wait for input Immediate return with EWOULDBLOCK error number (*select*() exception: READ)
Types of *write*() calls	Output buffers available Immediate return	Immediate return No output buffers available Wait for output buffers Immediate return with EWOULDBLOCK error number (*select*() exception: WRITE)
accept() call	New connection Immediate return	Immediate return No connections queued Wait for new connection Immediate return with EWOULDBLOCK error number (*select*() exception: READ)
connect() call	Wait	Immediate return with EINPROGRESS error number (*select*() exception: WRITE)



(a) by number of calls (b) by latency

(c) by number of calls (d) by latency (e) by average latency

FIGURE 4. Systems calls across the system.

TABLE 6. Comparision of I/O multiplexing operations.

System Call	select	poll	epoll
File descriptor limit	1024	No limit	No limit
Memory allocation	Fixed size array	Dynamically allocated array	Dynamically allocated tree
Event notification mechanism	Bitmaps of file descriptors	Separate fields of input and output events per-socket	Event-based
Scalability	Poor (linear)	Poor (linear)	Excellent (logarithmic)
Kernel memory usage	High (bitmap)	High (array)	Low (tree)
Portability	Widely available on most platforms	Widely available on most platforms	Linux-specific

blocking *read* and *write*, where a process will wait for data while on the CPU. Instead, one waits for the multiplexing I/O system calls to determine which files are ready to be read or written to.

Table 6 compares various types of I/O multiplexing system calls, with *epoll* demonstrating the best performance, while *select* and *poll* are better suited for small numbers of active connections [44], [45]. *epoll* offers two modes: level-triggered (LT) polling and edge-triggered (ET) polling. In the level-triggered mode, notifications are received whenever a file descriptor is ready, while in the edge-triggered mode, notifications are received whenever a change occurs. Suppose

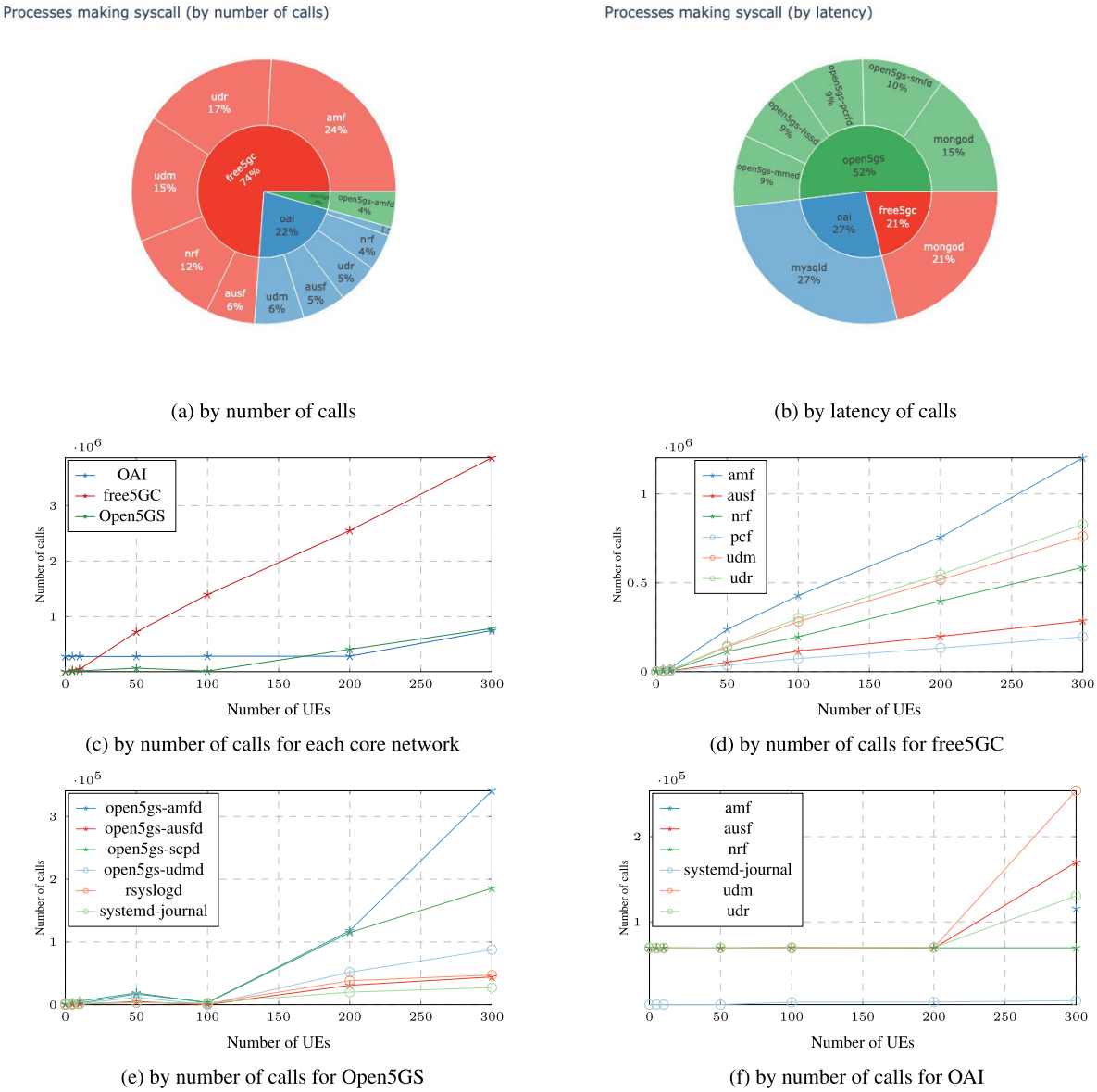


FIGURE 5. **Processes making system calls.**

that we have a buffer with 2KB of data, and only 1KB can be read at a time, LT and ET will both trigger when the 2 KB data is received. After reading the first 1KB, LT will trigger again, whilst ET will not trigger. Therefore, ET reduces the number of *epoll* system calls.

From the results in Figure 6a, we observe that free5GC and Open5GS have a significantly higher number of *epoll*/*poll*/*select* system calls than OAI. These figures show the sum of all the system calls made by all processes running on a core network. For more details on how the system call usage and frequency vary for each process, see Figures published on our webpage [20].

In certain cases, the high frequency of system calls can be unnecessary, which can impact the performance of the system. Some of the potential reasons for this include:

1) The number of ready file descriptors exceeds the *maxevents* parameter specified in the *epoll_wait*. This means that there are more files ready to be read from or written to than can be provided by a single *epoll_wait* system call. In this case, the system call iterates through them in a round-robin fashion. To verify this, one can compare the number of file descriptors provided by *epoll_wait* with the *maxevents* parameter. Using *bpftrace*, we developed a tool (epolldiff . bt) [20] to inspect two kernel tracepoints to values for *maxevents* and the ready file descriptors returned.
2) The presence of a signal handler that interrupts *epoll_wait*/ *poll*/ *select*, leading it to return with an error code (EINTR). It may be necessary to check for this error and retry the *epoll_wait* system call. To validate

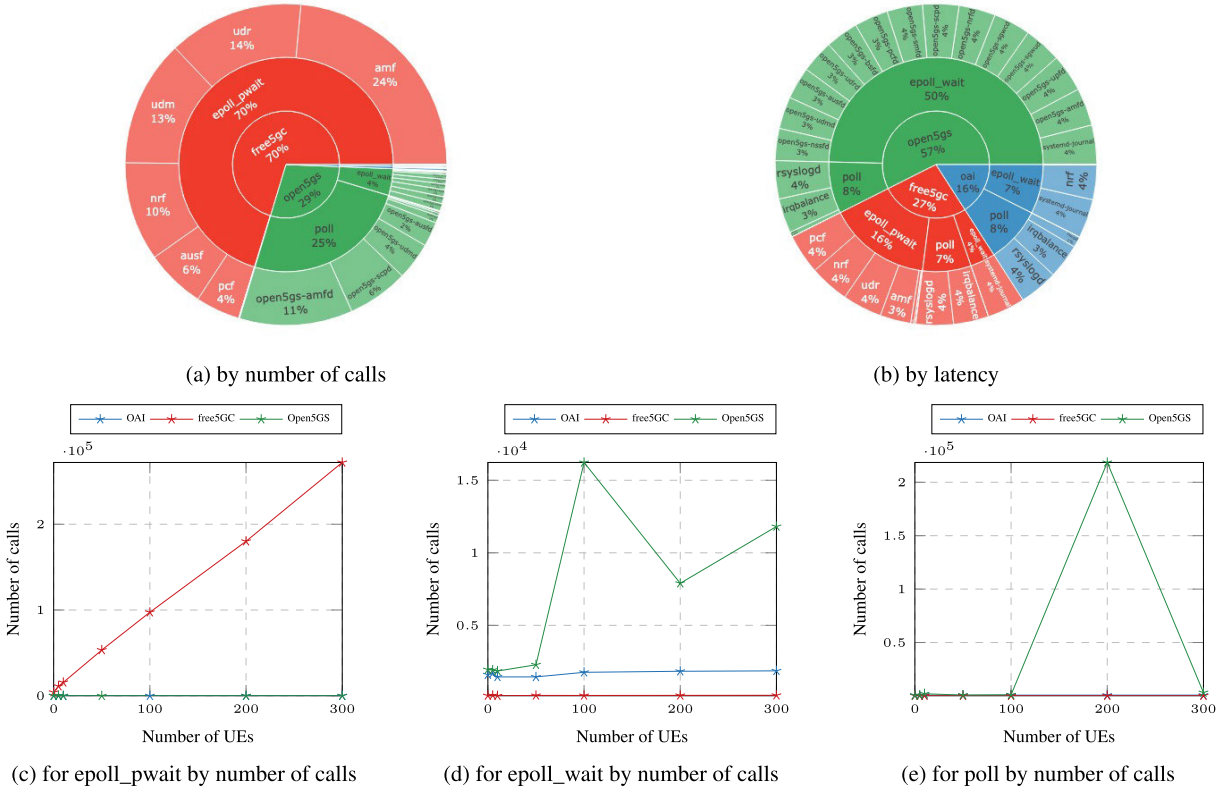


FIGURE 6. I/O Multiplexing systems calls across the system.

this, we executed the command in Listing 1, which generates a list of failed system calls.

3) The timeout value for *epoll_wait* is too low. To investigate this, we examine a histogram of the *epoll_wait* timeout and analyse the range of timeout values versus the ready events returned. By correlating these values with the number of returned values, we can determine if the duration of the timeout is a contributing factor. We provide a *bpftrace* tool epolltime . bt [20] to inspect the values and make the corresponding changes.
4) The buffer used to read data from ready sockets is too small. As a result, the *epoll_wait* keeps triggering because the available data have not been fully read. This can be resolved by running *epoll* in ET; we recommend the developers to run the *epoll* in ET mode. We can inspect the *epoll* file descriptors to check if it is in ET mode in two steps: (i) get the file descriptors for *epoll* and then (ii) check if the flags are set on the file descriptor. The commands to execute these actions are given in Listing 1. We can also obtain further information or confirmation by monitoring the respective system calls reading the file descriptors, e.g., *read*, *rcv* etc.

The OAI makes limited use of I/O multiplexing system calls. This means that it does not take advantage of the performance benefits from I/O multiplexing system calls. Since the core network involves communication between NFs and the 5G base station (gNB), without using I/O

```
# Get system calls returning error code
sudo python3 syscount.py -x --d 2

# Get the file descriptor for epoll_wait system
    calls
bpftrace -e 't:syscalls:sys_enter_epoll_wait  @pid
    [comm, pid, args->epfd] = count();'

# Check if \ac{ET} flag is set for a file
    descriptor
cat /proc/[pid]/fdinfo/[epfd]
```

LISTING 1. Syscalls returning errors

multiplexing, the OAI core network will have to use blocking system calls, as we will see in the later results sections. Although not the most performant, free5GC has the highest frequency of I/O multiplexing system calls, namely *epoll_pwait*. The free5GC developers should inspect further on the configurations and usage of the system call to ensure that it is not making unnecessary calls by following the steps described earlier. On the other hand, from the results in Figure 6a Open5GS calls the *poll* the most. As discussed above, *epoll* demonstrates the best performance, especially with high active connections. The Open5GS developers can verify that the system will not benefit from adopting *epoll* instead of *poll*.

4) NANOSLEEP/CLOCK_NANOSLEEP

The *nanosleep* and *clock_nanosleep* system calls are used to allow the calling thread to sleep for a specific interval with

nanosecond precision [46]. The *clock_nanosleep* differs from *nanosleep* in two ways. Firstly, it allows the caller to select the clock against which the sleep interval is to be measured. Secondly, it enables the specification of the sleep interval as either an absolute or a relative value. Using an absolute timer is useful to prevent timer drift issues mentioned about *nanosleep* [46].

The *clock_nanosleep* and *nanosleep* system calls cause the thread or process to give up the remainder of its time slice and enter the "Not Runnable" state for the specified duration. The system calls are mostly used in polling use cases, where a program goes to sleep to wait for specific events to occur and wakes up periodically to check if those events have taken place. If any events are detected, then the program automatically services them before going back to sleep. It is important to note that the system may resume the thread before the sleep time is complete, resulting in an error code of *EINTR*. Developers must check for this error code and recall *clock_nanosleep* again if necessary. This could be one of the reasons for more *clock_nanosleep* system calls than desired, as in this case in Figures 7a and 7b. Whenever the system resumes a thread before sleep time is complete, it would mean a minimum of two *clock_nanosleep* system calls. We can confirm this by examining the return codes of the system calls using the *BCC* tool *argdist* as depicted in Listing 2.

```
# Check if thread call sleep resumes before sleep time completes

# For nanosleep syscall
python3 argdist.py -C 't:syscalls:sys_exit_nanosleep():u16:args->ret' -i 5 -d 5

# For clock_nanosleep syscall
python3 argdist.py -C 't:syscalls:sys_exit_clock_nanosleep():u16:args->ret' -i 5 -d 5
```

LISTING 2. Inspect threads calling sleep syscall

From Figure 7, free5GC uses the sleep systems calls more than the other core networks. Furthermore, the usage of nanosleep by free5GC increases linearly with the increase in UEs. In general, sleep is used to wait for an event to occur or a process to finish. Ideally, as the number of events increases, more events should be waited on the same interval, thus increasing the number of sleep intervals with a decreasing rate. The free5GC developers should look into the possibility of bulk waiting or batch processing. On the other hand, proportioning the free5GC results in Figure 7a and Figure 7b, we can notice that *clock_nanosleep* has a higher latency compared to *nanosleep*. Therefore, unless Open5GS and OAI are using the *clock_nanosleep* specific features, they should consider making use of the *nanosleep* system call instead.

Figures 7 show the combined system call usage of all processes running on a core network. For more details on how the system call usage and frequency vary for each process, see Figures published on the project's webpage [20].

5) FUTEX

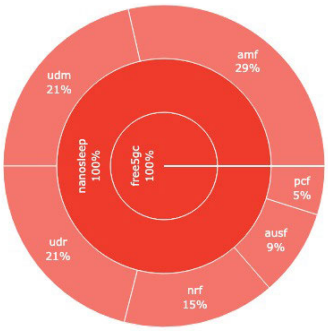
The *futex*() system call offers a mechanism to wait until a specific condition becomes true. It is typically used as a blocking construct in the context of shared-memory synchronisation. Additionally, *futex*() operations can be employed to wake up processes or threads that are waiting for a particular condition [47]. The main design goal of *futex* is to manage the *mutex* keys in the user space to avoid context switches when handling *mutex* in kernel space. In the *futex* design, the kernel is involved only when a thread needs to sleep or the system needs to wake up another thread. Essentially, the *futex* system call can be described as providing a kernel side wait queue indexed by a user space address, allowing threads to be added or removed from the user space [48]. A high frequency of calls to the *futex* system may indicate a high degree of concurrent access to shared resources or data structures by multiple threads or processes.

We can get further details on the usage of *futex* syscalls by tracing how they are being produced and the return value. This can tell us the source of the *futex* syscalls and the state of the syscalls when they return. One of the sources of *futex* calls is when locks are being used. The high *futex* calls may indicate lock contention. We obtain statistics on *mutex* locks by tracing the *mutex* events. Additionally, we can count *mutex*-related events in process threads (pthreads) to get an idea of how often the *mutex*es are being called. We use the *BCC* tool deadlock.py to get an idea of potential deadlocks from the locks. Lastly, we gain further insight by looking at the code path that leads to *futex* syscalls per process and further draw flame graphs for the code paths. The Listing 3 gives the commands that can be used to get the information discussed in this paragraph.

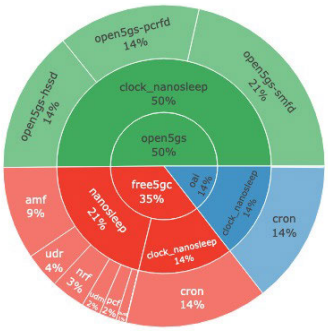
```
# Get return value for futex syscalls
trace.py -U -a 'r::sys_futex "%d", retval'

# Get stats on mutex
python3 klockstat.py -d 20

# Get mutex related events in a process
python3 funccount.py u:pthread:*mutex* -d 20

# Get code path leading to futex syscalls
stackcount.py -f -PU -D 20 futex > futex_codepath.txt
```

LISTING 3. Inspect usage of futex syscall

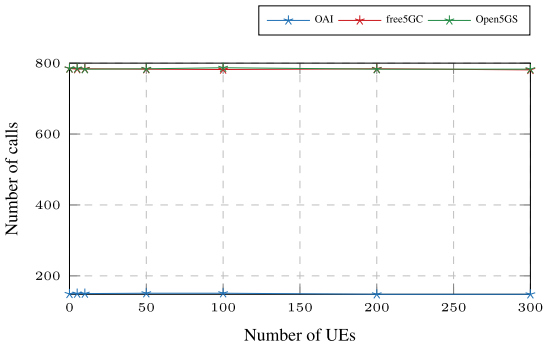
From Figure 8, we can see that free5GC uses the *futex* system call more than the other core networks. *Futex* helps in protecting concurrent access to resources. In general, access-locked resources affect the system performance; therefore, the results in Figure 8 show that the performance of free5GC is affected by resource contention. This supports the difference in performance between free5GC and Open5GS.



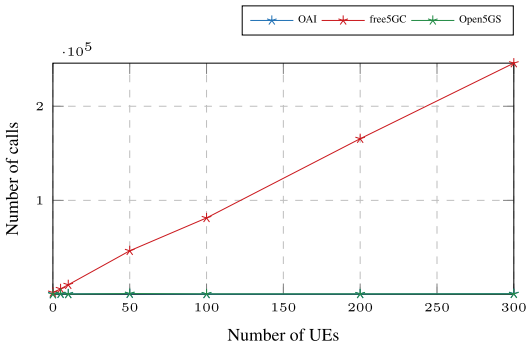
(a) by number of calls



(b) by latency



(c) for clock_nanosleep by number of calls



(d) for nanosleep by number of calls

FIGURE 7. Sleep systems calls across the system.

The results in Figure 8 show the aggregate system call usage of all processes running on a core network. For more details on how the system call usage and frequency vary for each process, please see Figures published on the project's webpage [20].

6) READ/WRITE

The *read*() system call [49] is used to retrieve data from a file stored in the file system, while the *write*() system call [50] is used to write data from a buffer to a file. Both system calls take into account the "count", which represents the number of bytes to read or write. Upon successful execution, these system calls return the number of bytes that were successfully read or written. By default, these system calls are blocking but can be changed to non-blocking using the *fnctl* system call. Table 5 describes the difference between blocking and non-blocking calls. Blocking is a problem for programs that should operate concurrently, since blocked processes are suspended. There are two different, complementary ways to solve this problem. They are nonblocking mode and I/O multiplexing system calls, such as *select* and *epoll* [51]. The architectural decision to use a combination of multiplexing I/O operations and non-blocking system calls offers advantages depending on the use cases. Some scenarios where this approach is beneficial include situations where small buffers would result in repeated system calls [51], when the system is dedicated to one function, or when multiple I/O system calls return an error [44].

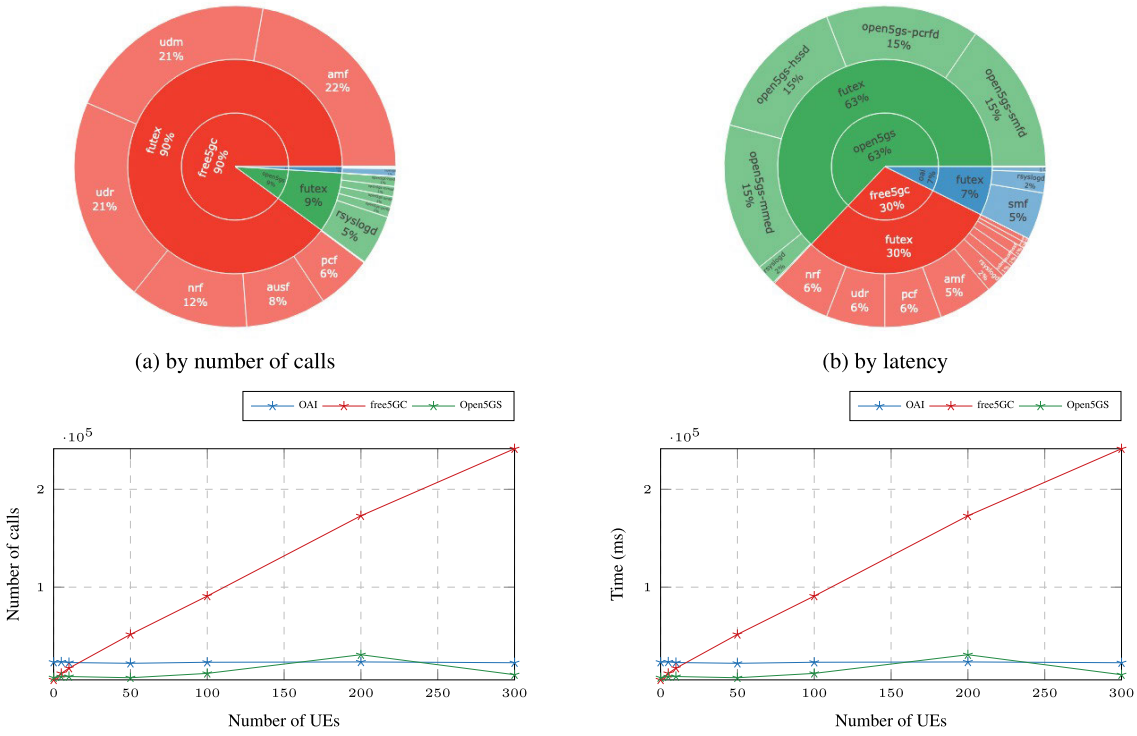
We can examine the size of the buffer and compare it with the number of bytes successfully read or written. Given this information, we can determine whether the buffer size is small. We can also check if the operations are interrupted, which would result in a higher frequency of the system calls. To do this, we wrote *bpftrace* scripts, *readinfo.bt* and *writeinfo.bt*, which gives the buffer size versus the bytes read or written per process and file descriptor.

From the results in Figure 9, we see that OAI makes the most use of the read/write system calls. To reduce the performance impact of the read/write system calls, the system calls should be used in conjuction with the I/O multiplexing system calls, and can optionally be set to non-blocking system calls. In the earlier sub-section VII-B3, we already saw that OAI does not make use of the I/O multiplexing system calls. Therefore, it would benefit from using I/O multiplexing system calls for better performance.

Figures 9 show the combined system call usage of all processes running on a core network.

7) RECV, RECVFROM, RECVMSG, RECVMMSG

recvfrom(), *recvmsg*() and *recvmmsg*() are all system calls used to receive messages from a socket. They can be used to receive data on a socket, whether or not it is connection-orientated. These system calls are blocking calls; if no messages are available at the socket, the receive calls wait for a message to arrive. If the socket is set to non-blocking, then the value -1 is returned, and *errno* is



(a) by number of calls

(b) by latency

(c) for futex by number of calls

(d) for futex by latency

FIGURE 8. Systems calls for resource contention across the system.

set to EAGAIN or EWOULDBLOCK [52]. Passing the flag MSG_DONTWAIT to the system call enables non-blocking operation. This provides behaviour similar to setting O_NONBLOCK with *fcntl* except MSG_DONTWAIT is per operation.

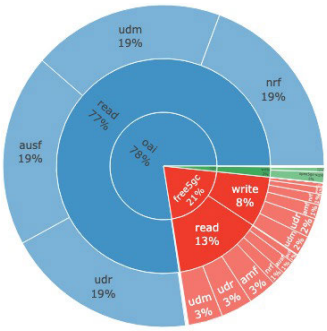
The *recv*() call is normally used only on a connected socket and is identical to *recvfrom*() with a nil from parameter.

To improve the performance of a system with too many *recv*() system calls, we can:

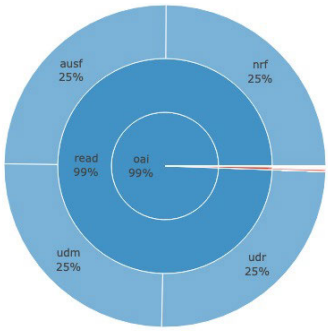
1) Read multiple messages in a single system call through *recvmmsg*. Notably, this may not result in significant improvements [53], [54], [55].
2) Use non-blocking sockets. By checking the number of *recv* system calls that returned 0, we can get an idea of the wasted blocking time. We use a simple *bpftrace* inline script to get the number of syscalls that return 0, for (i) *recvfrom* (ii) *recvmsg* and (iii) *recvmmsg*. We can also inspect the sockets when they are being created to see if they are set to non-blocking mode. To achieve this, we modified the *BCC* tool *argdist* to give us the name of the process creating the sockets. We can then use the modified *argdist* determine if the (i) client side sockets and (ii) server side sockets are set to non-blocking mode. The commands for obtaining the information are given in Listing 4.
3) Use I/O multiplexing system calls. These system calls allow one to wait for data to arrive on multiple sockets at once. The absence of I/O multiplexing will likely result in a large number of *recv* calls returning 0. We further inspect whether I/O multiplexing is being employed by printing the stack trace for the system calls *recvfrom*(), *recvmsg*() and *recvmmsg*().
4) Use a combination of non-blocking sockets and I/O multiplexing [51]

```
# Number of recvfrom syscalls that does return any data
bpftrace -e 't:syscalls:sys_exit_recvfrom { @pid[comm, pid, args->ret] = count(); }'

# Number of recvmsg syscalls that does return any data
bpftrace -e 't:syscalls:sys_exit_recvmsg { @pid[comm, pid, args->ret] = count(); }'

# Number of recvmmsg syscalls that does return any data
bpftrace -e 't:syscalls:sys_exit_recvmmsg { @pid[comm, pid, args->ret] = count(); }'

# Check if client sockets are set to non-blocking mode
argdist.py -c -C 't:syscalls:sys_enter_socket():int,int,int:$COMM,args->protocol,args->family&00004000'

# Check if server sockets are set to non-blocking mode
argdist.py -c -C 't:syscalls:sys_enter_accept4():int,int,int:$COMM,args->fd,args->flags&00004000'
```

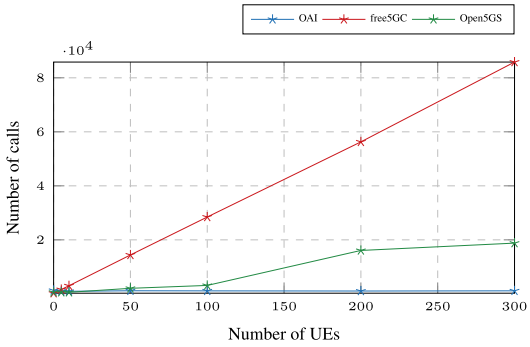
LISTING 4. Inspect usage of recv syscalls



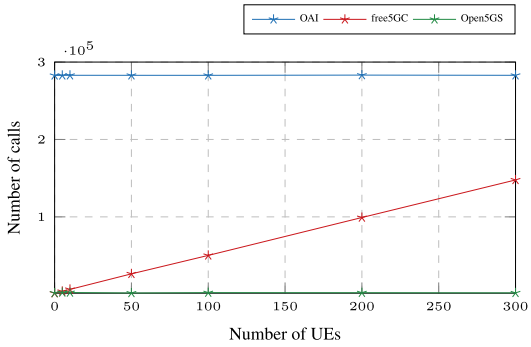
(a) by number of calls



(b) by latency



(c) for write by number of calls



(d) for read by number of calls

FIGURE 9. Systems calls for reading and writing to files across the system.

From the results in Figure 10a, we can see that Open5GS makes more receive system calls. Proportionally comparing the performance of free5gc and Open5GS, Open5GS likely has room for improvement in regards to the receive system calls. We also see from Figure 10a and Figure 10b that some of the system calls are made by process that are not part of the registration or de-registration procedure. Therefore, Open5GS can reduce the number of system calls by enabling the deployment of the core network with network functions relevant to the usage scenarios. The figure shows the aggregated results for all the processes active on a system call.

8) SEND, SENDTO, SENDMSG, SENDMMSG

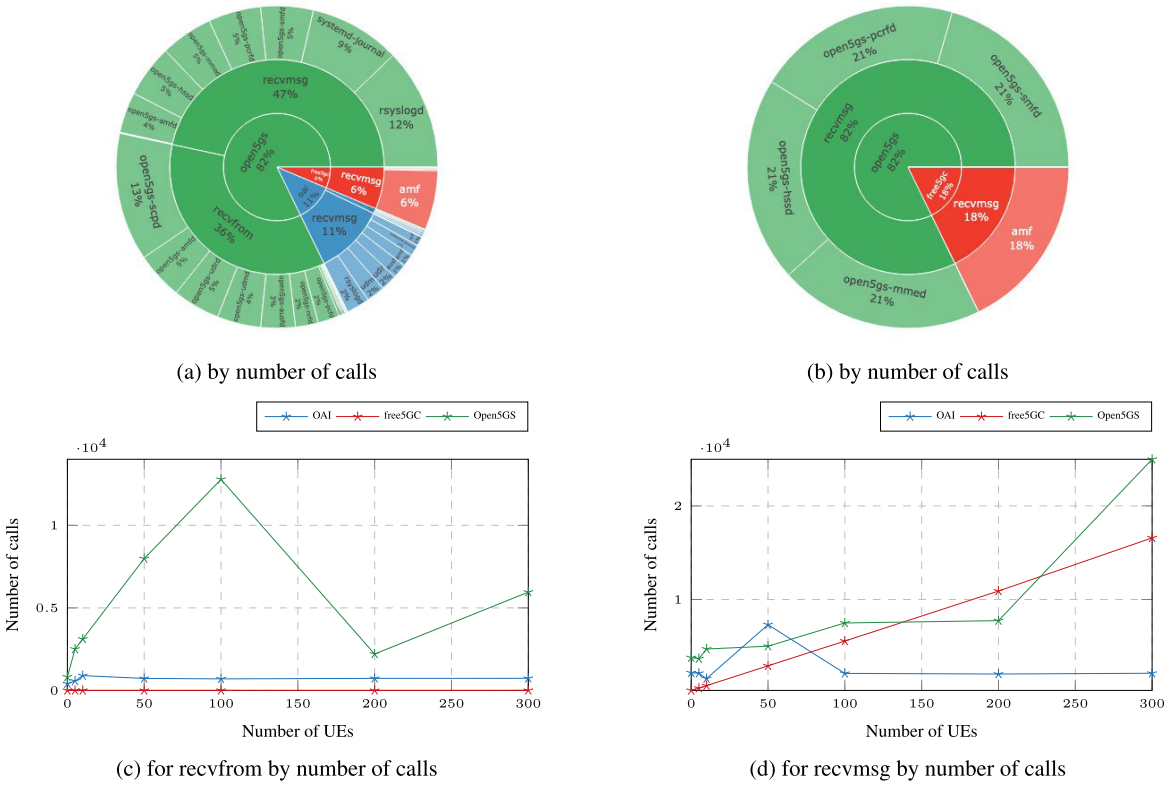
The *send*() call may only be used when the socket is in a connected state (so that the intended recipient is known). The *send*() is similar to *write*() with the difference of flags. The *sendto* and *sendmsg* work on both connected and unconnected sockets. The *sendmsg*() call also allows sending ancillary data (also known as control information) [56].

The approaches to optimise the *send*(s) system calls are similar to the discussed approaches for the *recv*(s) system calls. These include I/O multiplexing, using the system calls in non-blocking mode, and sending multiple messages in a single system call where possible.

From the results in Figure 11, we can see that OAI has the most *send* system calls called on sockets. The information is important for developers to look further into how the system calls are used. On the other hand, we see that Free5gc has a linear increase of the *send* system call. Depending on how the system call is being used, Free5gc can potentially benefit from batch send system calls, and I/O multiplexing where it is not being used.

9) SCHED_YIELD

The *sched_yield* system call is used by a thread to allow other threads a chance to run, and the calling thread relinquishes the CPU [57]. Strategic calls to *sched_yield*() can improve performance by giving other threads or processes an opportunity to run when (heavily) contended resources, such as *mutexes*, have been released by the caller [57]. The authors of [58] were able to improve the throughput of their system by employing the *sched_yield* system call after a process processes each batch of packets before calling the *poll*. On the other hand, *sched_yield* can result in unnecessary context switches, which will degrade system performance if not used appropriately [57]. The latter is mainly true in generic Linux systems, as the scheduler is responsible for deciding which process runs. In most cases, when a process yields, the scheduler may perceive it as a higher priority and still put it back into execution, where it yields again in a loop. This behaviour is mainly due to the algorithm and logic used by Linux's default scheduler to determine the process with the higher priority, as explained in the forum [59] by Linus Torvalds. That is, on a generic Linux system, the use of *sched_yield* often implies unnecessary system calls and indicates poor design or legacy software.



(a) by number of calls

(b) by number of calls

(c) for recvfrom by number of calls

(d) for recvmsg by number of calls

FIGURE 10. Systems calls for socket receive across the system.

From the results in Figure 12c, we see that it is majorly free5GC which makes use of the sched_yeid system call. It also affects the performance of a system. The free5GC developers need to look at the necessity of the *sched_yield* system call. In the case that it is legacy software, the developers should consider upgrading it, and in the case that it is not, the developers should consider removing it.

VIII. DISCUSSION

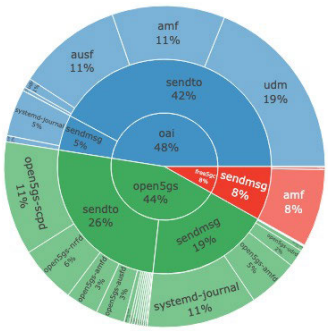
Our study and analysis reveal three key factors that significantly contribute to system performance as measured by micro-benchmarking: (i) the rate at which a particular system call increases as the load or number of UEs (User Equipment) increases, (ii) the total number of system calls, and (iii) the time it takes to execute a system call, which we also refer to as the latency of system calls.

The rate at which a system call increases can be interpreted as the gradient of the line that connects the number of calls versus the number of UEs. For multiplexing system calls like *epoll_wait* and wait system calls like *nanosleep*, a higher gradient is undesirable, as it indicates excessive resource usage. For blocking system calls, such as *write* or *recvmsg*, a gradient significantly greater than 1 suggests that system calls increase disproportionately to the increase in load or UEs, potentially affecting performance.

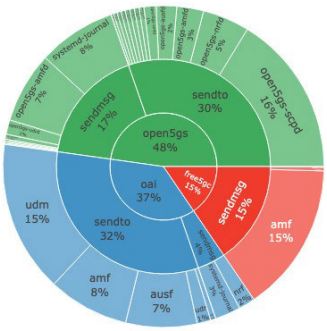
An excessive number of system calls generally degrades system performance. However, this depends on the type of system call. A high number of multiplexing system calls may actually save the need for an even higher number of blocking system calls, such as *recvmsg* or *read*. Therefore, the interpretation of call numbers should be context-dependent, considering the specific system call.

The time required to execute a system call contributes to the latency of a process or operation. A slow system call can cause delays in processing, affecting overall performance. Processes can use alternative system calls, such as nanosleep and *clock_nanosleep*, to optimise performance. In addition, underlying issues or implementation problems can lead to excessive system call latency. This is exemplified by the case of futex system calls, which exhibited high latency due to inherent limitations.

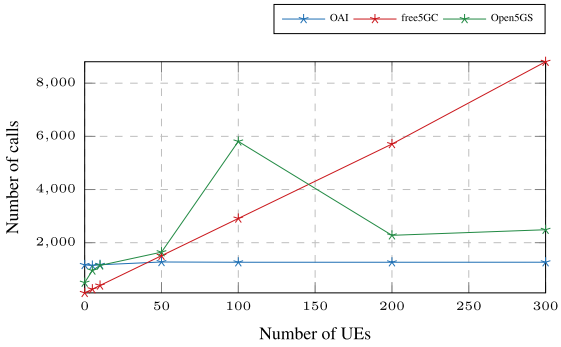
Figures 13a and 13b provide a summary of system call usage patterns. The x-axis represents the cumulative number of calls for a specific system call across various UE counts. The y-axis indicates the rate of increase in the number of calls as the number of UEs increases. In Figure 13a, each bubble represents a particular system call. Conversely, in Figure 13b, each bubble represents a specific process. In both cases, the radius of the bubbles corresponding to the latency of the system calls. By comparing the macro-benchmarking results of the overall system in Figure 1 with the micro-benchmarking summary, we can observe a correlation. Open5GS, the most performant core network according to macro-benchmarking, exhibits system calls concentrated in the lower left quadrant of Figure 13a, and



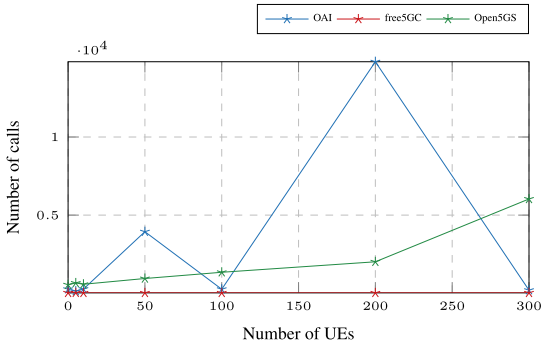
(a) by number of calls



(b) by number of calls

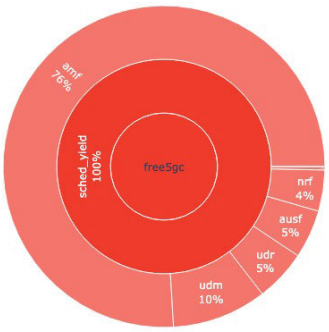


(c) for sendmsg by number of calls

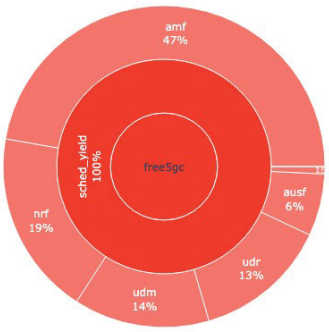


(d) for sendto by number of calls

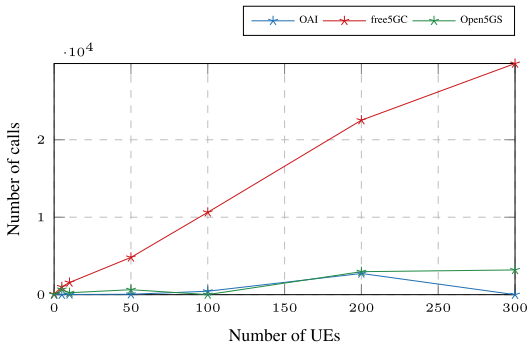
FIGURE 11. Systems calls for socket send across the system.



(a) by number of calls



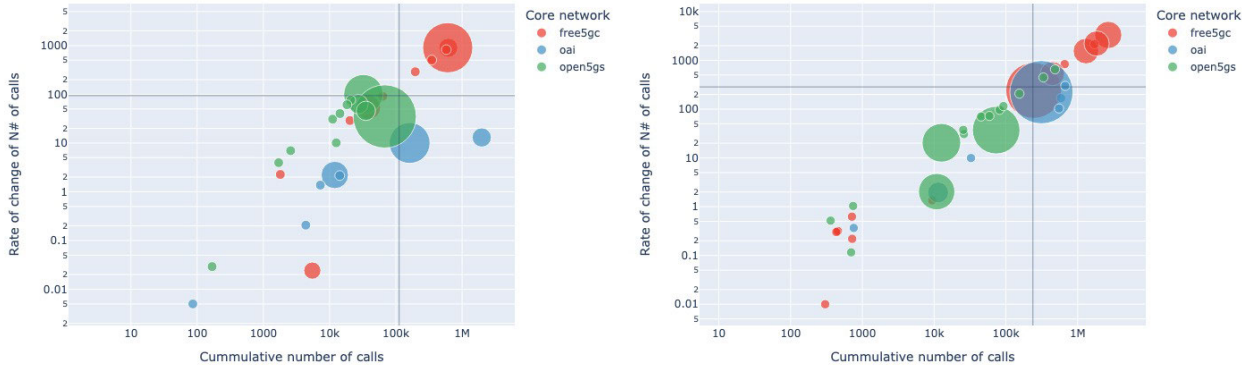
(b) by number of calls



(c) for sched_yield by number of calls

FIGURE 12. Systems calls for sched_yield across the system.

most of its processes, with the exception of two, reside in the lower left quadrant of Figure 13b. This implies that, compared to other core networks, Open5GS's software architecture scales its underlying resources proportionally or with a lower



(a) by syscalls

(b) by processes

FIGURE 13. System resource usage patterns.

proportion to the increase in UEs. In other words, it uses the same resources for new UEs or less. Furthermore, compared to other core networks, it initiates system resource usage (through system calls) less frequently.

Comparison of free5GC and OAI reveals that both have processes and system calls in the right quadrants, indicating a high number of system calls. As stated previously, the higher number of calls should be interpreted differently depending on the system call. Although both have elevated numbers, the axis is in logarithmic form, and the magnitude difference between processes or system calls at the edge of the axis is proportionally larger than depicted. OAI issues the blocking *read* system call twice more frequently than free5GC issues its system calls in the right quadrants. This magnitude difference aligns with the observed performance disparities between these two systems. Additionally, free5GC utilises multiplexing system calls in the right quadrant, which, as previously discussed, can indicate resource conservation.

Despite the performance differences observed among the core networks, which are linked to their underlying resource usage patterns through system calls, all core networks can enhance their resource utilisation by implementing the recommended strategies and conducting further analysis, as described in the results section. Our detailed analysis and actionable recommendations offer a valuable contribution to the advancement of open-source 5GC, benefiting the broader open-source community. These improvements hold the potential to enhance the accumulation of network users, for example, during mobility management scenarios.

IX. CONCLUSION

This study presents a framework to evaluate the feature support of 5G Core (5GC) networks. We assess the feature support against the Network Function (NF) operations and 5G procedures from Release 17. The resulting framework produces a wealth of artefacts detailing the supported 5G NF operations and the resulting 5G procedures that are fully or partially supported. Due to the large number of results generated, they have been made available on a website. This framework for feature support assessment can assist researchers and network administrators in selecting a 5GC that meets their research or deployment needs and in evaluating the maturity of open-source implementations of 5GCs.

In addition to the feature support assessment, this study provides a comprehensive evaluation of the runtime performance of open-source 5G core networks during UE registration and de-registration, with a particular focus on control plane network functions. By assessing the performance of the 5GC implementations using macro-benchmarking and micro-benchmarking approaches, this study provides insight into the software architectural design of each 5GC and identifies areas for performance improvement. Based on these evaluations, the study provides recommendations for software architectural changes that can minimise performance degradation resulting from the virtualisation of 5GC networks. These recommendations can be implemented by both open-source projects and 5GC vendors to optimise their 5G core network solutions.

The results indicate that 5GC implementations can improve their design by optimising their usage of I/O multiplexing system calls. Additionally, the implementations can improve their use of shared memory and locks to reduce resource contention on shared memory resources or lock contention. The study also identified the use of *sched_yield* system call by free5GC, which on Linux platforms may be unnecessary and/or indicate the use of legacy software. Within the limits of our study, the time taken to register a given set of UEs increases linearly with the number of UEs registered. This can be improved by redesigning the usage of I/O multiplexing and using non-blocking system calls, among other approaches detailed in the study. Overall, the performance difference among the 5GCs can be correlated with their relative usage patterns and choice of system calls. The study proposes that by implementing the recommendations and suggestions in this study, the performance gap between the core networks can be reduced. Furthermore, this study incorporates comprehensive tutorials and related source codes to enable replication of the benchmarking experiments.

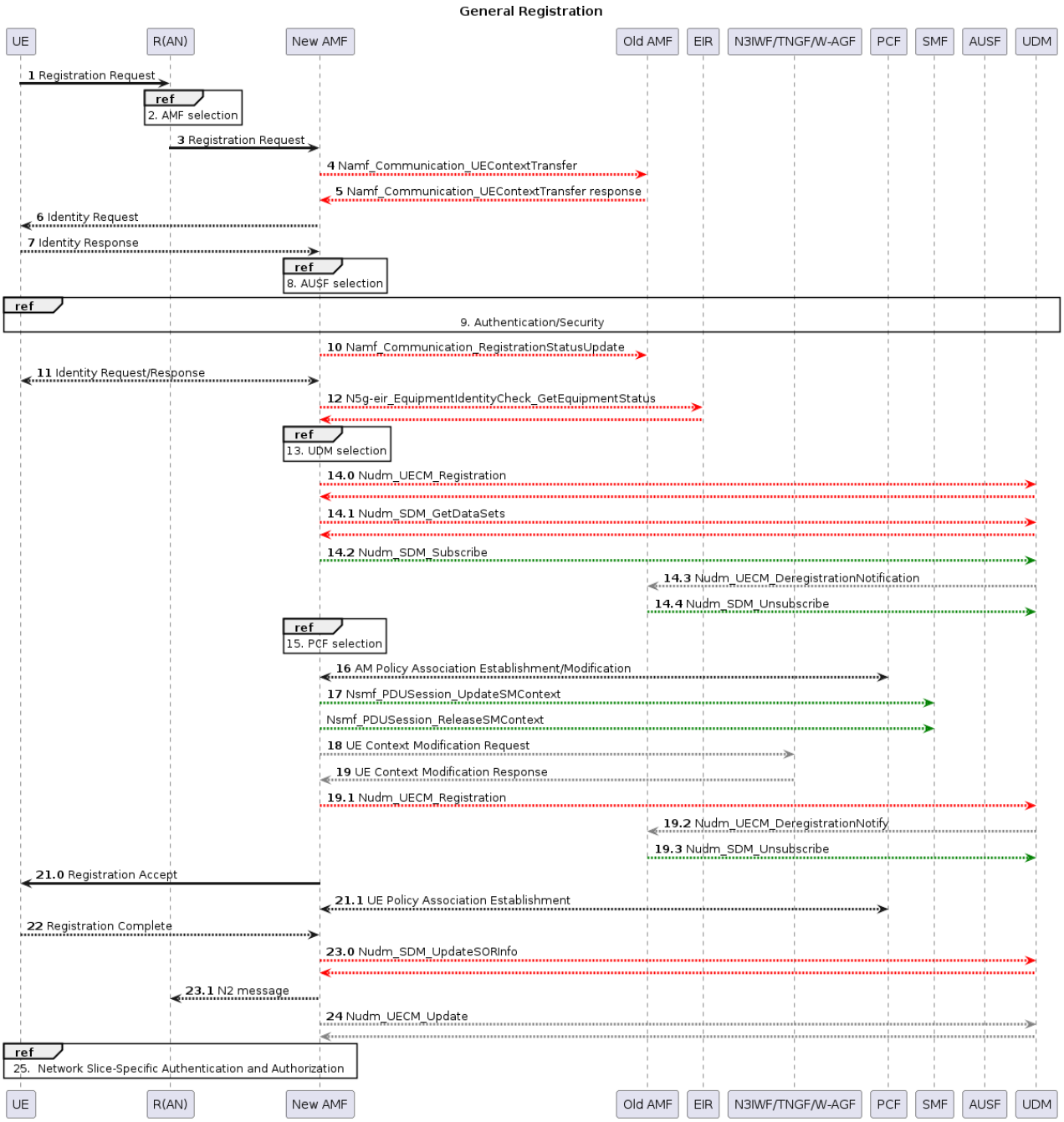


FIGURE 14. Open5GS: UE Registration procedure.

Complementing the recommendations, this study delves into the relationship between macro-benchmarking results, which reflect overall system performance, and micro-benchmarking results, which illuminate the underlying resource utilization patterns of the core networks. The discussion section examines the distribution and usage patterns of system calls to explain the observed performance discrepancies among the core networks.

One important area of future work is to include macro and micro benchmarking as part of a continuous integration (CI) pipeline that runs on major software changes. This will help to ensure that performance regressions are detected early and that new features do not introduce new performance bottlenecks.

Another important area of future work is to align the micro-benchmarking procedure with the standard specification for benchmarking, for example, Network Function Virtualization (NFV) Release 3; Testing; Specification of Networking Benchmarks and Measurement Methods for NFVI. This will make the results of the benchmarking more

FIGURE 15. free5GC: UE Registration procedure.

comparable to other studies and more useful to the 5GC community.

Furthermore, it is important to evaluate the other control plane-related procedures on top of the current registration and de-registration procedures evaluated in the study. This will provide a more complete picture of the performance of 5GC implementations and help to identify other areas where improvements can be made. Finally, in the next steps, we will compare the Open5GCore from FOKUS with the other implementations.

APPENDIX A
REGISTRATION PROCEDURES

In table 3, we show the NF service operations that are supported or not supported by each core network. The diagrams in figs. 14 to 16 show the procedure flows for registration, highlighting the operations that are supported or not supported by the core network. This gives us an indication of the capabilities that can be fulfilled by each core network per procedure. We maintain a list of diagrams for the Release 17 procedures on the GitHub repository [29], where we aim

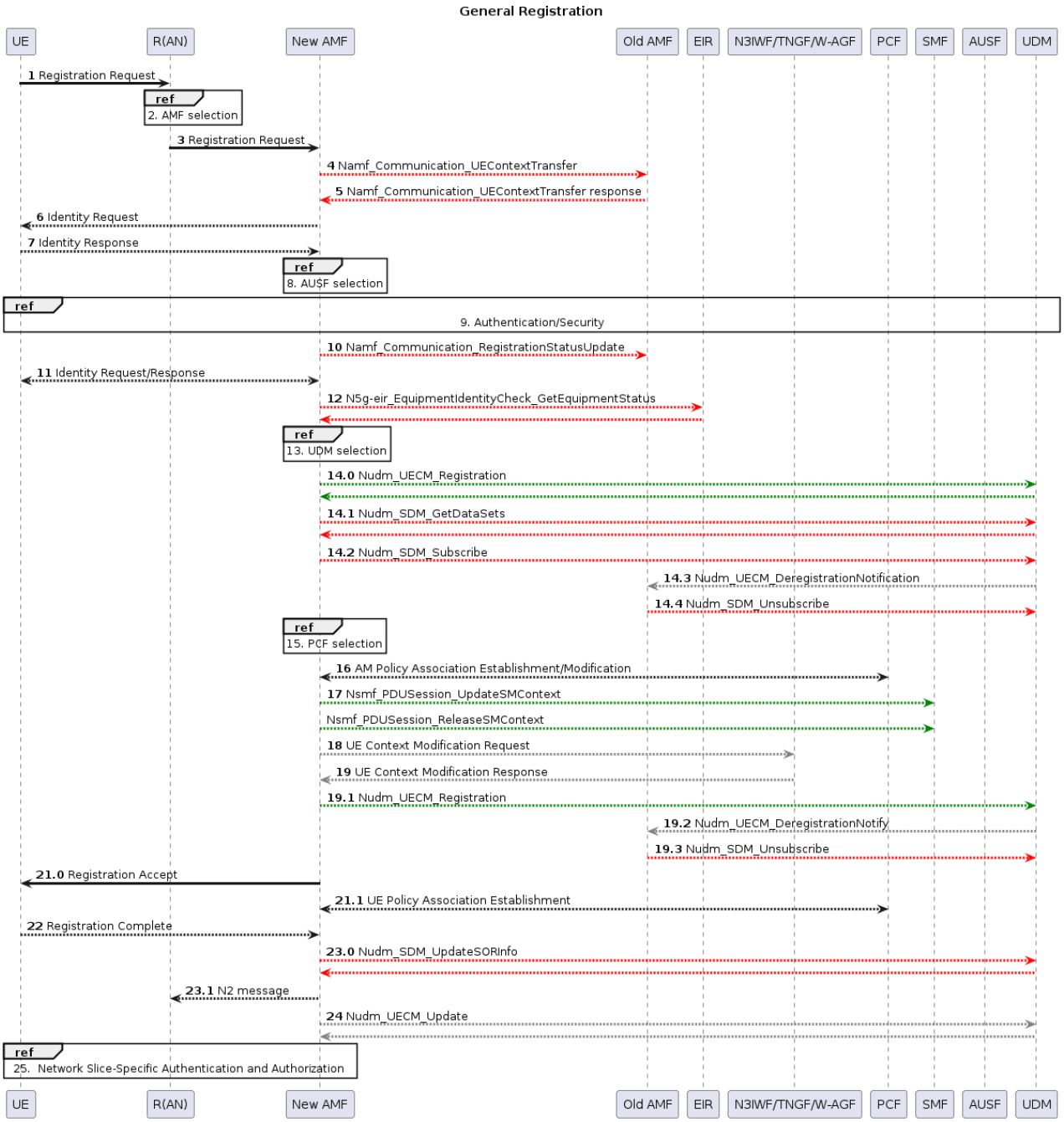


FIGURE 16. OAI: UE Registration procedure.

to continuously update it with the help of the community. The colour keys for the diagram are as follows:

- **Black** depicts NAS or NG Application Protocol (NGAP) operations which are not evaluated in this study
- **Green** depicts operations supported by the core network on the given procedure
- **Red** depicts operations that are not supported by the core network per a given procedure
- **Grey** depicts operations we were not able to evaluate whether they are supported or not

REFERENCES

[1] *ETSI*, Standard 3GPP TS 23.501, 2018.
[2] (2020). *3GPP TS 38.401 ETS*. Accessed: Jun. 21, 2023. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/15.02.00_60/ts_123501v150200p.pdf
[3] M. Ahmad, S. U. Jafri, A. Ikram, W. N. A. Qasmi, M. A. Nawazish, Z. A. Uzmi, and Z. A. Qazi, "A low latency and consistent cellular control plane," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., architectures, protocols Comput. Commun.*, Jul. 2020, pp. 648–661.
[4] N. Van Tu, J.-H. Yoo, and J. W. Hong, "EVNF–Hybrid virtual network functions with Linux eXpress data path," in *Proc. 20th Asia–Pacific Netw. Operations Manage. Symp. (APNOMS)*, Sep. 2019, pp. 1–6.

[5] J. Mwangama and N. Ventura, "Accelerated virtual switching support of 5G NFV-based mobile networks," in *Proc. IEEE 28th Annu. Int. Symp. Pers., Indoor, Mobile Radio Commun. (PIMRC)*, Oct. 2017, pp. 1–7.
[6] H. Zhang, Z. Chen, and Y. Yuan, "High-performance UPF design based on DPDK," in *Proc. IEEE 21st Int. Conf. Commun. Technol. (ICCT)*, Oct. 2021, pp. 349–354.
[7] J. Rischke, C. Vielhaus, P. Sossalla, J. Wang, and F. H. P. Fitzek, "Comparison of UPF acceleration technologies and their tail-latency for URLLC," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2022, pp. 19–25.
[8] T. A. N. do Amaral, R. V. Rosa, D. F. C. Moura, and C. E. Rothenberg, "An in-kernel solution based on XDP for 5G UPF: Design, prototype and performance evaluation," in *Proc. 17th Int. Conf. Netw. Service Manage. (CNSM)*, Oct. 2021, pp. 146–152.
[9] A. H. Vasoukolaei, D. Sattar, and A. Matrawy, "TLS performance evaluation in the control plane of a 5G core network slice," in *Proc. IEEE Conf. Standards Commun. Netw. (CSCN)*, Dec. 2021, pp. 155–160.
[10] V. Jain, H.-T. Chu, S. Qi, C.-A. Lee, H.-C. Chang, C.-Y. Hsieh, K. K. Ramakrishnan, and J.-C. Chen, "L 2 5GC: A low latency 5G core network based on high-performance NFV platforms," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 143–157.
[11] OpenAirInterface. (2023). *5G Core Network*. Accessed: Apr. 22, 2023. [Online]. Available: https://openairinterface.org/oai-5g-core-network-project/
[12] Nat. Chiao Tung Univ. (2023). *Free5GC*. Accessed: Apr. 22, 2023. [Online]. Available: https://www.free5gc.org/
[13] (2023). *Open5GS*. Accessed: Apr. 22, 2023. [Online]. Available: https://open5gs.org/
[14] Fraunhofer FOKUS. (2023). *Open5gcore*. Accessed: Apr. 22, 2023. [Online]. Available: https://www.open5gcore.org/
[15] A. B. Brown, "A decompositional approach to computer system performance evaluation," Harvard Comput. Sci. Group, Cambridge, U.K., Tech. Rep. TR-03-97, 1997, pp. 1–77.
[16] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang, "The case for application-specific benchmarking," in *Proc. 7th Workshop Hot Topics Operating Syst.*, 1999, pp. 102–107.
[17] L. Thomeczek, A. Attenberger, J. Kolb, V. Matousek, and J. Mottok, "Measuring safety critical latency sources using Linux kernel eBPF tracing," in *Proc. 32nd Int. Conf. Archit. Comput. Syst.*, May 2019, pp. 1–8.
[18] T. Mukute. (2023). *5G Core Network Traffic Generator*. [Online]. Available: https://github.com/tariromukute/CoreNetworkTrafficGenerator
[19] S. Shende, "Profiling and tracing in Linux," in *Proc. Extreme Linux Workshop*, vol. 2, 1999, pp. 1–26.
[20] T. Mukute. (2024). *Performance Benchmarking of 5G Core Networks*. [Online]. Available: https://tariromukute.github.io/control_plane_performance_analysis
[21] (2019). *3GPP Release 15*. Accessed: Apr. 22, 2023. [Online]. Available: https://www.3gpp.org/specifications-technologies/releases/release-15
[22] *System Architecture for the 5G System*, Standard TS 23.501, 3GPP, 2023.
[23] O. O. Erunkulu, A. M. Zungeru, C. K. Lebekwe, M. Mosalaosi, and J. M. Chuma, "5G mobile communication applications: A survey and comparison of use cases," *IEEE Access*, vol. 9, pp. 97251–97295, 2021.
[24] *System Architecture for the 5G System (5GS)*, Standard TS 123 501, V17.8.0, 2023.
[25] *Procedures for the 5G System (5GS)*, Standard TS 123 502, v17.8.0, 2023.
[26] *System; Restoration Procedures*, Standard TS 123 527, V17.6.0, 2021.
[27] *Telecommunication Management; Charging Management; 5G System; Services, Operations and Procedures of Charging Using Service Based Interface (SBI)*, Standard TS 132 290, V17.6.0, 2021.
[28] *Security Architecture and Procedures for 5G System*, Standard TS 133 501, V15.16.0, 2021.
[29] T. Mukute. (2023). *Completeness Evaluation of 5G Core Networks*. [Online]. Available: https://github.com/tariromukute/5gc-features
[30] J. Struye, B. Spinnewyn, K. Spaey, K. Bonjean, and S. Latre, "Assessing the value of containers for NFVs: A detailed network performance study," in *Proc. 13th Int. Conf. Netw. Service Manage. (CNSM)*, Nov. 2017, pp. 1–7.
[31] R. Rizki, A. Rakhmatsyah, and M. A. Nugroho, "Performance analysis of container-based Hadoop cluster: OpenVZ and LXC," in *Proc. 4th Int. Conf. Inf. Commun. Technol. (ICoICT)*, May 2016, pp. 1–4.
[32] I. O. Ploten, "Zero copy packet processing," M.S. thesis, Fac. Inf. Technol., Brno Univ. Technol., Brno, Czechia, 2019.
[33] W. Mauerer, *Professional Linux Kernel Architecture*. Hoboken, NJ, USA: Wiley, 2010.
[34] L. Yu, "Operating system process management and the effect on maintenance: A comparison of Linux, freebsd, and Darwin," *INFOCOMP J. Comput. Sci.*, vol. 5, no. 2, pp. 38–44, 2006.
[35] B. Gregg, *BPF Performance Tools*. Reading, MA, USA: Addison-Wesley, 2019.
[36] W. Stallings and G. K. Paul, *Operating Systems: Internals and Design Principles*, vol. 9. New York, NY, USA: Pearson, 2012.
[37] A. Zafeiropoulos, E. Fotopoulou, M. Peuster, S. Schneider, P. Gouvas, D. Behnke, M. Muller, P.-B. Bök, P. Trakadas, P. Karkazis, and H. Karl, "Benchmarking and profiling 5G verticals' applications: An industrial IoT use case," in *Proc. 6th IEEE Conf. Netw. Softwarization (NetSoft)*, Jun. 2020, pp. 310–318.
[38] S. M. Pieper, J. M. Paul, and M. J. Schulte, "A new era of performance evaluation," *Computer*, vol. 40, no. 9, pp. 23–30, Sep. 2007.
[39] B. Gregg. (2015). *Choosing a Linux Tracer*. [Online]. Available: https://brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html
[40] T. Oberdörfer, "Characterization of interrupt handling in board management controllers," B.S. thesis, Dept. Comput. Sci., ETH Zurich, Zurich, Switzerland, 2021.
[41] J. Pavela, "Efficient techniques for program performance analysis," M.S. thesis, Fac. Inf. Technol., Brno Univ. Technol., Brno, Czech Republic, Jul. 2020. [Online]. Available: https://www.fit.vut.cz/study/thesis/19092/
[42] (2023). *UERANSIM*. [Online]. Available: https://github.com/aligungr/UERANSIM
[43] L. B. D. Silveira, H. C. de Resende, C. B. Both, J. M. Marquez-Barja, B. Silvestre, and K. V. Cardoso, "Tutorial on communication between access networks and the 5G core," *Comput. Netw.*, vol. 216, Oct. 2022, Art. no. 109301.
[44] Sobyte. (2022). *Epoll—Efficiently Handling Large Number of File Descriptors*. Accessed: Apr. 2, 2023. [Online]. Available: https://www.sobyte.net/post/2022-04/epoll-efficiently/
[45] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, "Comparing and evaluating epoll, select, and poll event mechanisms," in *Proc. Linux Symp. 2004*, 2004, pp. 1–20.
[46] M. Kerrisk. (2021). *Clock Nanosleep(2) Linux Manual Page*. Accessed: Apr. 1, 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/clock_nanosleep.2.html
[47] M. Kerrisk. (2021). *Futex(2)—Linux Manual Page*. Accessed: Apr. 1, 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/futex.2.html
[48] A. Almeida. (2022). *Landing a New Syscall: What is Futex?*. [Online]. Available: https://www.collabora.com/news-and-blog/blog/2022/02/08/landing-a-new-syscall-part-what-is-futex/
[49] M. Kerrisk. (2022). *Read(2)—Linux Manual Page*. Accessed: Apr. 1, 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/read.2.html
[50] M. Kerrisk. (2022). *Write(2)—Linux Manual Page*. Accessed: Apr. 1, 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/write.2.html
[51] E. Eklitzke. (2012). *Blocking I/O, Nonblocking I/O, and Epoll*. [Online]. Available: https://eklitzke.org/blocking-io-nonblocking-io-and-epoll
[52] M. Kerrisk. (2023). *Recv(2)—Linux Manual Page*. Accessed: Apr. 4, 2023. [Online]. Available: https://manpages.ubuntu.com/manpages/bionic/man2/recv.2freebsd.html
[53] G. Ara, L. Lai, T. Cucinotta, L. Abeni, and C. Vitucci, "A framework for comparative evaluation of high-performance virtualized networking mechanisms," in *Proc. 10th Int. Conf.*, 2020, pp. 59–83.
[54] M. Sojka, P. Písa, and Z. Hanzálek, "Performance evaluation of Linux CAN-related system calls," in *Proc. 10th IEEE Workshop Factory Commun. Syst. (WFCS)*, May 2014, pp. 1–8.
[55] I. Marinos, "Network and storage stack specialisation for performance," Ph.D. dissertation, Fac. Comput. Sci. Technol., Univ. Cambridge, Cambridge, U.K., 2018.
[56] M. Kerrisk. (2021). *Send, Sendto, Send Message–Send a Message on a Socket*. Accessed: Apr. 14, 2023. [Online]. Available: https://www.man7.org/linux/man-pages/man2/send.2.html
[57] M. Kerrisk. (2022). *Sched_Yield–Yield the Processor*. Accessed: Apr. 10, 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/sched_yield.2.html
[58] F. Huici, C. R. E. Matus, G. Tsolis, C. Pisa, S. Salsano, F. Lombardo, N. Blefari-Melazzi, G. Bianchi, L. Krug, P. Veitch, P. Eardley, T. Kanceki, E. Curley, J. Thomson, M. Flouris, A. Nanos, X. Ragiadakos, J. Chesterfield, K. Du, and L. Toms, "Mechanisms for network service dynamics and performance," Consorzio Nazionale Interuniversitario per le Telecomunicazioni, Parma, Italy, DELIVERABLE D5.2, 2018.
[59] L. Torvalds. (2020). *No Nuances, Just Buggy Code (Related to Spinlock Implementation and the Linux Scheduler)*. Accessed: Apr. 14, 2023. [Online]. Available: https://www.realworldtech.com/forum/?threadid=189711&curpostid=189752

TARIRO MUKUTE is a rising Researcher in network performance. He tackles the challenge of optimising virtualised 5G platforms for enhanced network function performance. His Ph.D. work with the University of Cape Town is focussing on designing and implementing a dynamic 5G N6-LAN, with added focus on improving performance. He has authored several publications and open-source code contributions. Driven by interests in computer and mobile networks, 5G core, and future internet technologies, he actively contributes to testbeds and open-source projects, translating his research into practical solutions.

LUSANI MAMUSHIANE received the master's degree in electrical engineering, specializing in software-defined wide area networks (SDWAN) from the University of Cape Town (UCT), where she is currently pursuing the Ph.D. degree.

In her role as a Senior Researcher with the Council for Scientific and Industrial Research (CSIR), she specialises in programmable networks, including open EPC and open RAN, leveraging cutting-edge technologies, such as software-defined networking (SDN) and network function virtualization (NFV). Additionally, her expertise extends to pivotal domains, such as cloud computing, the Internet of Things (IoT), and artificial intelligence (AI). Her research focuses on the intersection of machine learning with 5G and beyond network slicing.

ALBERT A. LYSKO (Senior Member, IEEE) received the Ph.D. degree from the Norwegian University of Science and Technology, Norway, in 2010. He was with both academia and industry, in Europe and Africa. Currently, he is a Principal Researcher with the Council for Scientific and Industrial Research (CSIR), South Africa. While at CSIR, his leading experimental research in television white spaces (TVWS) provided internet to over 20,000 users in three countries and enabled setting up the South African National TVWS regulation and contributed to TVWS regulations in other African countries and USA. He has authored three patents, a book, two book chapters, and over 150 research articles, popular science, and news articles. His research interests include numerical electromagnetics, smart antennas, dynamic spectrum access, and 5G and 6G. He is a fellow of the South African Institute of Electrical Engineers. He holds three best paper and several professional awards. He has several IEEE awards for volunteering.

ELENA-RAMONA MODROIU received the master's degree in computer science from Politehnica University Bucharest, Romania, in 2003, and the Postgraduate degree from the Universitat Politècnica de València, Spain. She is a Researcher with the Technische Universität Berlin (DE). She has been actively involved in VoIP/SIP co-founding the Kamailio open-source project and the SIP server at the core of IMS solutions for 4G/5G open-source based deployments. Her research work covering the fields of 5G mobile networks and end-to-end setups, with a focus on core networks and the evolution from 5G toward 6G generation of mobile communications.

THOMAS MAGEDANZ (Senior Member, IEEE) has been the Director of the business unit software-based networks (NGNI), Fraunhofer Institute for Open Communication Systems FOKUS (https://www.fokus.fraunhofer.de/go/ngni), Berlin, since 2003. He has also been a Professor with the Technische Universität Berlin, Germany, where he has been leading the Chair for Next Generation Networks (https://www.av.tu-berlin.de), since 2004. For 33 years, he has been a globally recognized ICT expert, working in the convergence field of telecommunications, internet, and information technologies understanding both the technology domains and the international market demands. His current research interests include software-based networks for different verticals, with a strong focus on public and non-public campus networks and the evolution from 5G to 6G.

JOYCE MWANGAMA (Member, IEEE) received the B.Sc. degree in electrical and computer engineering and the M.Sc. and Ph.D. degrees in electrical engineering from the University of Cape Town, South Africa, in 2008, 2011, and 2017, respectively. She is currently an Associate Professor with the Department of Electrical Engineering, University of Cape Town. She has published her research work in several peer-reviewed publications. Her research work has also contributed to the universities for future internet and the testbeds for reliable smart city machine-to-machine communications international research collaboration projects.