Published at “3rd International Conference on Future Internet of Things and Cloud”, © 2015 IEEE DOI 10.1109/FiCloud.2015.77

Performance analysis of OpenAirInterface system emulation

Antonio Virdis, Niccolò Iardella, Giovanni Stea
Dipartimento di Ingegneria dell’Informazione
University of Pisa, Italy

Dario Sabella
Telecom Italia Lab, Turin, Italy

Abstract—**With the rapid growth of mobile networks, the radio access network becomes more and more costly to deploy, operate, maintain and upgrade. The most effective answer to this problem lies in the centralization and virtualization of the eNodeBs. This solution is known as Cloud RAN and is one of the key topics in the development of fifth generation networks. Within this context OpenAirInterface is a promising emulation tool that can be used for prototyping innovative scheduling algorithms, making the most of the new architecture. In this work we first describe the emulation environment of OpenAirInterface and its scheduling framework and we use it to implement two MAC schedulers. Moreover we validate the above schedulers and we perform a thorough profiling of OpenAirInterface, in terms of both memory occupancy and execution time. Our results show that OpenAirInterface can be effectively used for prototyping scheduling algorithms in emulated LTE networks.**

Keywords—LTE; OpenAirInterface; Cellular Networks; performance evaluation; 5G.

I. Introduction

One of the most critical components of a cellular network is the radio access network (RAN), i.e. the network base stations (eNB in the LTE terminology) which provides coverage and connectivity to the users; it is the main asset of a network operator and, above all, the subject of the largest share of investments. A new, emerging approach in the road towards the 5th generation of cellular networks (5G) is to split the functions of the eNBs into two main parts (Fig. 1): a first one that deals with radio access and a second one that copes with the digital and base band functions [1]. The former is called Remote Radio Head (RRH) and is deployed in the territory generally according to coverage policies. The latter is called Base Band Unit (BBU) and can be centralized and remotely connected with the RRH.

The centralization of BBUs opens new perspectives in the management of computational power for cellular networks. In the last years a new paradigm, known as Cloud RAN (C-RAN), has been proposed [2]. The main idea is to create a pool of BBUs that can be allocated dynamically, depending on the load of the system. When C-RAN is implemented in a virtualized environment (virtual RAN), BBUs are managed as virtual machines, that are instantiated on demand onto physical machines. This allows operators to manage in a smarter way power and processing resources, for example by turning off unloaded cells and assigning more resources to heavily loaded ones.

From a computer engineering perspective, the C-RAN approach paves the way to several research opportunities: centralization of access network processing will allow the implementation of smart algorithms for cooperation among the base stations in terms of radio resource allocation, QoS-aware traffic management and power saving. In fact, in a traditional RAN, the effectiveness - if not the viability itself - of these algorithms is hindered by the fact that eNBs have limited computation power, and communicate using latency-prone, bandwidth-constrained interfaces. It is thus hardly possible to convey to a central decision point all the information required to make a global decision for several cells, even more so given that scheduling decisions are made at millisecond timescales. In a C-RAN, instead, all the information pertaining to a large number of neighboring cells (e.g., the QoS subscription profile and the channel and buffer state of users) can be conveyed to a central processing cloud, where inter-eNB communication becomes *inter-process* communication, and a relatively large amount of processing resources (i.e., number of cores and memory) can be harvested on demand for computation-intensive tasks. Parallel to revising the corpus of well-known research problems typical of 4G cellular networks, in order to adapt them to such a totally different environment, C-RAN opens new, interesting research directions: specifically, how to design virtualization frameworks which are flexible and agile so as to meet the challenging constraints of a 5G network. For instance, how to allocate computational resources to virtualized BBUs so as to allow radio resource allocation algorithm to work properly, how to setup and tear down BBUs for underloaded cells dynamically, transferring their state seamlessly to other BBUs, and what algorithms to use to make this happen.

Among the recent research initiatives in that direction, the Flex5GWare EU H2020 project [14] – which involves the authors of this paper - aims at building cost-effective hardware/software platforms for 5G so as to increase the hardware versatility and reconfigurability, increase capacity and decrease the overall energy consumption.

The common factors that influence the end-user quality of service (QoS) are the amount of bandwidth available for transmission (i.e., resource blocks, RBs) and the requested data rates. In the context of C-RAN, the new factor that arises is the computational capacity on the BBU pool side. The amount of resources that are needed by a BBU to run have a direct impact on how the C-RAN system scales. This leads on side to a careful evaluation of the capacity of a BBU pool, and to the development of efficient algorithms for BBU management on the other.

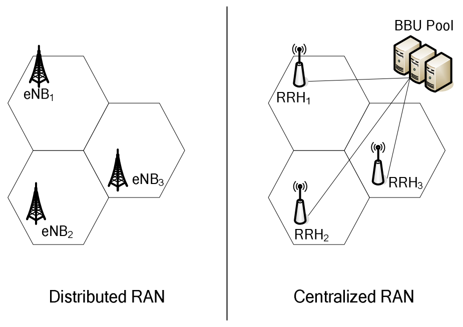

Fig. 1. Comparison of Distributed and Centralized RAN architecures.

In order to carry out research on these topics, the first requirement is to have a software platform capable of emulating cellular networks on general purpose hardware, but at the same time to implement accurately the LTE protocol stack. Although system level simulators (e.g. SimuLTE [4]) give good solutions for large-scale testing of algorithms, they fail to capture the effects of *computational load* of LTE systems. In recent years several *emulation* projects are instead being proposed. Leaving aside closed-source, commercial products (e.g. Amarisoft [3]) for obvious reasons, among open-source solutions Eurecom's OpenAirInterface (OAI) appears to be the most promising and complete project, especially by virtue of its emulation capacity, which allows one to conduct performance evaluation campaigns on LTE networks with a protocol stack entirely implemented in software [7]. In addition, OAI allows one to carry out experiments using hardware equipment and commercial terminals, by ensuring real-time performances and simplifying the passage from validation to prototyping.

OAI is already being used as a platform for experimentation and prototyping, hence has been the subject of previous studies, mainly concerned on evaluating the impact of single components, such as the abstraction of the physical layer and the number of users ([5] and [6]). However, to the best of our knowledge, aside from some recent publications [15] no thorough profiling of the code has been performed so far.

In this work, we give a twofold contribution. First, we put to the test OAI system-level emulation, assessing its viability as a testbed for prototyping algorithms . We analyse the OAI MAC-scheduling structure and we show how to implement and integrate new schedulers into it. Our analysis shows that OAI can be used to implement, test and validate MAC schedulers in a fully emulated environment, closer to real-world implementations than network simulation software. Second, we profile the OAI code, uncovering some non-intuitive results. First, the system is memory-hungry, and memory usage is strongly related to the number of users. Second, traffic generation has little impact on the performance of the system. Third, even the lightweight physical layer claimed to be useful to speed up emulation eats up most of the processing time. With respect to that, we propose some simple code optimizations that provide a speedup greater than two.

The rest of this paper is organized as follows. We first give an overview of OAI in section II. We then show in section III how to implement and validate resource-scheduling algorithms into the OAI framework. A comprehensive profiling of OAI is performed in section IV, analysing the impact of its components on system performance, from both a running time and memory usage perspective. Finally section V concludes the paper.

II. Overview of OpenAirInterface

OpenAirInterface is an open-source experimentation and prototyping platform for wireless communication systems, released under GPLv3 by Eurecom's Mobile Communications Department. Its main objective is to provide methods for protocol validation, performance evaluation and pre-deployment system test for LTE (Rel-8) and LTE-Advanced (Rel-10). The OpenAirInterface platform can be used for link-level simulation, system emulation and real-time Radio Frequency experimentation. It comprises the entire LTE protocol stack, including standard-compliant implementations of the 3GPP LTE access stratum for both eNB and UE and a subset of the 3GPP LTE evolved packet core protocols, and can be adopted as an emulation and performance evaluation platform for LTE/LTE-A systems [5].

The two main features of OAI are:

• An open implementation of a 4G system, fully compliant with 3GPP LTE standards and comprising the entire protocol stack, which can interoperate with commercial terminals and can be used for real-time experimentation.

• A built-in emulation capability which can be used to test repeatable scenarios in a software-only environment, and to switch *seamlessly* between emulation and real-time experimentation. For this purpose, two physical-layer (PHY) emulation modes are supported.

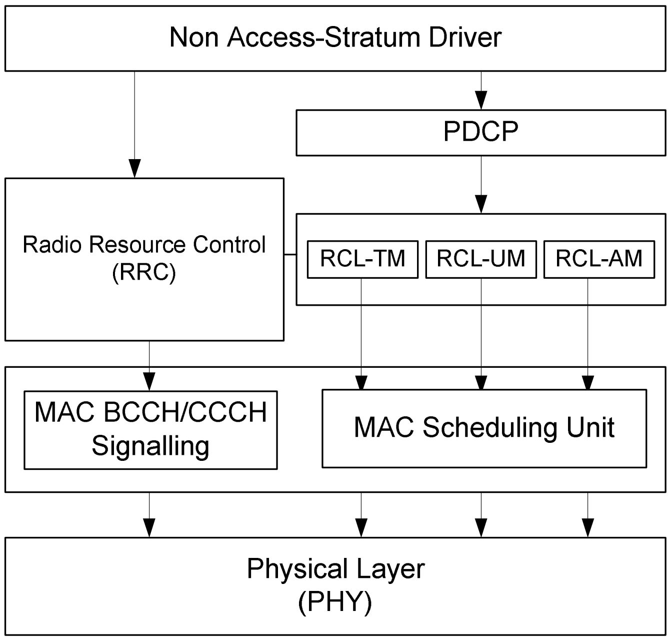

Fig. 2. The OAI protocol stack

TABLE I. MAIN OAI FEATURES

Supported releases	Rel-8.6, part of Rel-10
Duplexing modes	FDD, TDD
Carrier bandwidths	5, 10 and 20 MHz
Tx modes	1 (SISO), 2, 4, 5 and 6 (MIMO 2x2)
DL Channels	All
UL Channeles	All
HARQ support	

The implementation of the RAN provides the full protocol stack (PHY, MAC, RLC, PDCP, RRC) (see Fig. 2) for both eNB and UE, standard S1AP and GTP-U interfaces to the CN, IPv4 and IPv6 support, and a priority-based MAC scheduler which will be analysed in further detail later on. The main PHY features are summarized in TABLE I. .

OAI has a custom software-defined radio (SDR) front-end, called the ExpressMIMO2 PCIe board. The board features a small FPGA-based system and four high-quality RF chipsets which operate as MIMO front-ends for small cell eNBs, and they can be configured in frequency division duplexing (FDD) or time division duplexing (TDD) operation with channel bandwidths up to 20 MHz. Such system is completely open, at both the hardware and the software level. Moreover, OAI also supports USRP B210 platform via the UHD [8].

On the other hand, OAI can be used to emulate a complete LTE network [5], using the **oaisim** package. Several eNBs and UEs can be virtualized on the same machine or in different machines communicating over an Ethernet-based LAN. The PHY and the radio channels can be fully emulated or approximated with the so-called *PHY abstraction mode*. Note that this is not a simple simulation, but an emulation which runs the full protocol stack (in both PHY modes), using the same MAC code as the real-time application. This makes it possible to use the **oaisim** package to test sample scenarios (e.g., in a preliminary validation), without the uncertainties of the SDR frontend, and then switch seamlessly to the real-time environment.

OAI includes a traffic generator [9] (OTG), which can be mounted on top of the LTE stack and used to run an emulation with different loads. The generator includes several predefined traffic profiles, such as device-to-device (D2D), gaming, video streaming and full buffer, and can be customized using OAI scenario descriptors (OSDs).

III. IMPLEMENTING RESOURCE SCHEDULING ALGORITHMS

The MAC scheduler assigns transmission resources, i.e. RBs, to UEs. The actual amount of data that can fit into a RB depends on the channel quality perceived by the UE. For the purpose of DL scheduling, the UE reports a Channel Quality Indicator (CQI), i.e. a value between 0 and 15. The eNB uses it to decide the Modulation and Coding Scheme (MCS) to be used for communication with that UE, hence – indirectly – how many bytes will fit one RB. An exemplary mapping is reported in TABLE II.

A scheduling algorithm runs on every TTI and makes two decisions, either jointly or separately: first, how to *sort* UEs in a priority list, and – second – *how many* RBs to give to the UE being considered when cycling through that priority list. These decisions can be made based on several criteria, such as the CQI reported by the UEs – either the most recent or the average over a window, its backlog, the last time that UE was allocated RBs, its traffic profile, etc. We can categorize scheduling algorithms into two families: *stateless* and *stateful*. The former make decisions based only on the information available at the current TTI (e.g., the last reported CQI and current backlog of each UE), whereas the latter carry scheduling information across multiple TTIs.

TABLE II. EXEMPLARY CQI MAPPING

CQI	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bytes	0	3	3	6	11	15	20	25	36	39	50	63	72	80	93	93

In the rest of this section we will first present the OAI scheduling framework, aiming for a general view, then we will show how to implement and integrate new schedulers – both stateless and stateful – within it, using well-known schedulers from the literature, namely the Maximum-Carrier-over-Interference scheduler (MaxC/I) and the round-robin scheduler, as a proof of concept.

A. The OAI Scheduling Framework

The architecture of the OAI scheduling framework is undocumented. For this reason, we describe *where* scheduling takes place, and *how* it is done. The main operations of the OAI scheduling framework are carried out within the `eNB_dlsch_ulsch_scheduler()` function. The latter gets called every TTI and follows the callgraph shown in Fig. 3. The actual functions that are called depend on the TTI number and on the duplexing mode in use. Each of them is responsible for different operations, namely:

- `schedule_RA()` implements the random access scheduling;
- `schedule_ulsch()` implements the UL scheduling;
- `schedule_ue_spec()` manages the Hybrid ARQ (H-ARQ) procedures for packet re-transmission, schedules RBs to UEs according to the selected algorithm;
- `fill_DLSCH_dci()` composes the scheduling map in the Downlink Shared Channel (DLSCH) to let UEs know where to find their allocated RBs.

The scheduling algorithm is implemented into the `eNB_dlsch_scheduler_pre_processor()` function. The latter takes as an input the number N of available RBs to be allocated to the I active UEs, and the number Req_i of RBs required by each UE i.

OAI comes with a built-in stateful scheduler, which we call *legacy* scheduler for want of a better name. It consists of five main steps, shown in the left part of Fig. 4 and explained hereafter:

1. prepare the scheduling data structures;
2. compute a cap on the requested RBs for each UE, $\overline{RB} = N/I$;
3. allocate $RB_i = min\left\{Req_i, \overline{RB}\right\}$ to UE i ;
4. sort UEs according to several criteria: last TTI at which the UE was scheduled, current CQIs, backlog, etc.. The exact sorting is quite convoluted and is reported in Appendix A for completeness.
5. loop through the sorted list of UEs and allocate the remaining RBs to each UE that is still backlogged, so as to clear its backlog, until no more resources are available.

The above algorithm aims at ensuring fairness among UEs, by giving a minimum amount of resources to each back-logged one (step 3), and exploits spectral efficiency (step 4-5, specifically the part that considers CQIs).

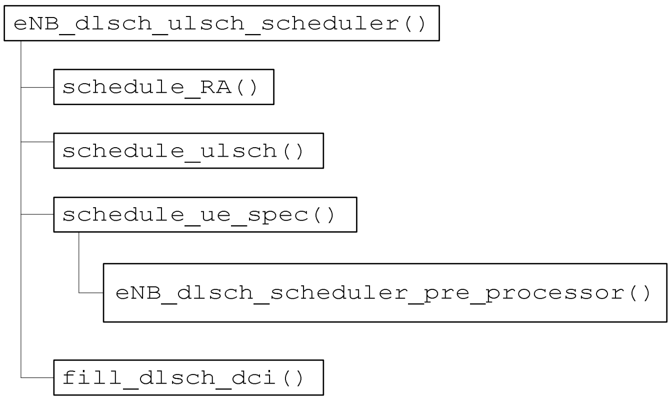

Fig. 3. Call graph for scheduling function

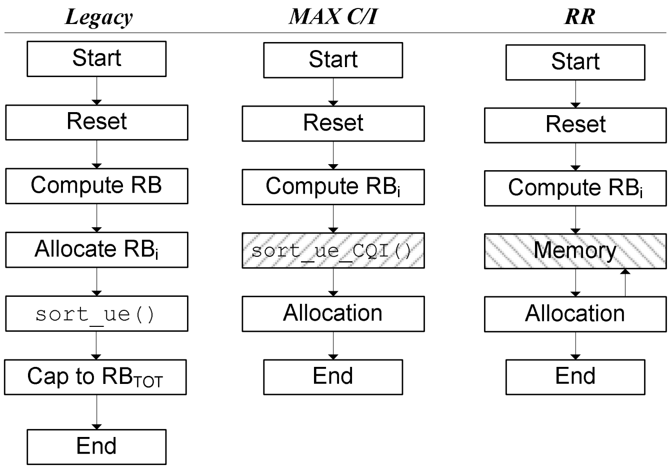

Fig. 4. Main operations for the legacy, MaxC/I, and RR scheduling algorithms, implemented in the eNB_dlsch_scheduler_pre_processor() function.

B. *Implementing New Scheduling Algorithms*

We implemented two well-known scheduling algorithms into the OAI scheduling framework. The first one, MaxC/I, is a stateless algorithm that maximizes the cell throughput by sorting UEs by decreasing CQI and serving each one exhaustively. The second one, Round Robin (RR), is stateful and ensures fairness by allocating a fixed number of RBs, called *quantum*, to each backlogged UE. Backlogged UEs are visited in a fixed order, hence a list pointer is saved at the end of a TTI and re-used in the next.

The main operations performed by either algorithm are shown in the right part of Fig. 4. We followed the same structure of the OAI *legacy* scheduler, adding new functions (marked in the figure) and modifying the existing ones when needed. The initialization is the same as for the *legacy* scheduler. The next two functions of the *latter* are then bypassed by simply computing the number of RBs requested by each user as $RB_i = Req_i$. From this point on the two schedulers start behaving differently.

MaxC/I sorts UEs by decreasing CQI. Thus, we implemented the `sort_ues_CQI()` function, using the same skeleton of the `sort_ue()` of the *legacy* scheduler. Scheduling is accomplished by allocating up to RB_i RBs to each UE, until enough resources are available.

RR instead stores the next UE to be served in a *memory* pointer. It then allocates up to *quantum* RBs to it and passes to the next one. The same operations are repeated until either no more resources are available or no UE is backlogged. Finally the id of next UE to be scheduled is stored in *memory* in order to be known in the next TTI.

It is clear from the above description that any scheduling algorithm – of either family – can be implemented via minor modifications once one knows where functions are located.

C. *Scheduler validation*

The above schedulers have been validated in a test scenario. For each UE we measure the average user- and cell-throughput at the application level, and the per-TTI average number of RBs allocated to each UE. We then use these values to compare the behavior of the schedulers. UEs are static and placed at a distance between 425 and 850 meters to the eNB

(Fig. 6). This configuration allows us to both impose different channel conditions among UEs and to test the schedulers behavior against a predictable outcome. The emulator is fed using a CBR traffic generator [9], which allows us to select both the packet size and the inter-packet time. A new packet is generated each millisecond for each target UE. Packets are composed of a payload, a header that mirrors higher-layer protocols, and additional control information (OTG Info) inserted by OAI for traffic statistics (Fig. 5). The main parameters used in the emulation are summarized in TABLE III. All the remaining ones assume default values.

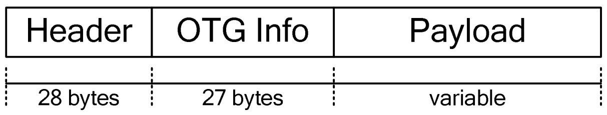

Fig. 5. Packet format for the OAI traffic generator

TABLE III. EMULATION PARAMETERS USED FOR PERFORMANCE EVALUATION

Parameter	**Value**
Emulation duration	10000 TTIs
Duplexing mode	FDD
Phy abstraction	Yes
# eNBs	1
# UEs	1 to 5
Mobility	Static
Traffic type	CBR
Payload Size	From 50 to 200 Bytes
RR *quantum*	2 RBs
# runs	9
# available RBs	25
eNB radiation pattern	Isotropic

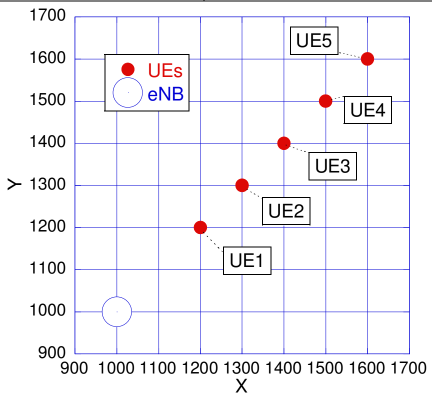

Fig. 6. System deployment for scheduling evaluation.

Published at "3rd International Conference on Future Internet of Things and Cloud", © 2015 IEEE DOI 10.1109/FiCloud.2015.77

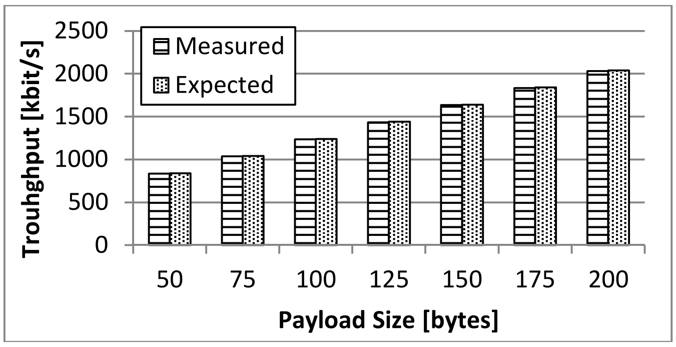

Fig. 7. Average user throughput with 1 UE and various payload sizes.

First of all we validate the system by measuring the throughput of a single UE for increasing payload size. In Fig. 7 we show that the results obtained with emulation are practically overlapping to the ones obtained using theoretical computations. The latter are computed as:

$$tpt = \left(p + h_{upper} + h_{OTG}\right) \times 1000 \times 8,$$

where p, h_{upper} and h_{OTG} are the payload, upper-layer header and OTG header size respectively.

In Fig. 8 we show instead the average number of RBs allocated to each UE with the three schedulers. UEs are at increasing distance to the eNB with UE 1 being the closest one. As we can see MaxC/I prioritizes UEs with higher CQIs, starving far away ones, e.g. UE 5. RR instead ensures airtime fairness by assigning the same number of RBs to each UE. Finally the *legacy* scheduler behaves similarly to RR, applying a soft prioritization based on CQIs. The RB allocation affects directly the average user throughput, as shown in Fig. 9.

In Fig. 10 we show the average cell throughput for an increasing number of UEs. The three schedulers exhibit the same behavior until saturation is reached (with three UEs). From that point on, MaxC/I achieves the same, and highest, cell throughput, since it always fits more bytes per RB. RR and the *legacy* scheduler instead achieve lower results, as they devote more airtime to UEs that are farther from the eNB. Each added node is in fact placed at greater distance (see Fig. 6), thus experiences a lower CQI. Finally, with 5 UEs, the *legacy* scheduler performs better than RR. This is due to the fact that it does exploits CQIs, albeit not exclusively (see the description of the sorting process in the Appendix), hence tends to fit more bytes than RR into one RB.

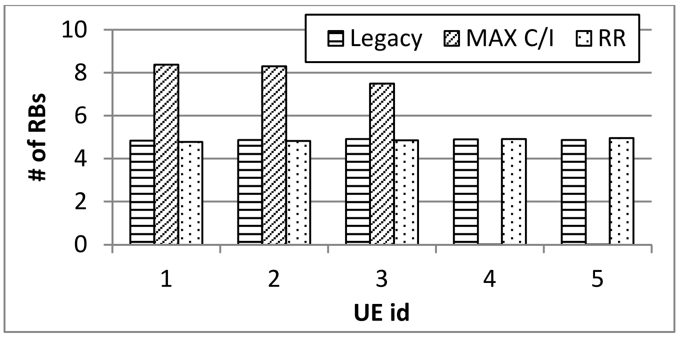

Fig. 8. Average RBs allocation with three scheduling algorithms and CBR traffic with a payload size of 150 Bytes.

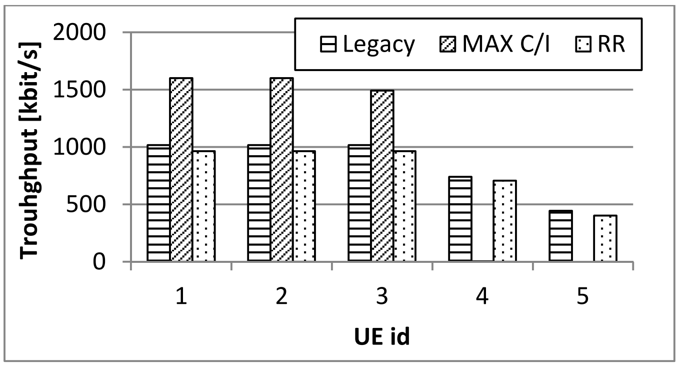

Fig. 9. Average user throughput with three scheduling algorithms and CBR traffic with a payload size of 150 Bytes.

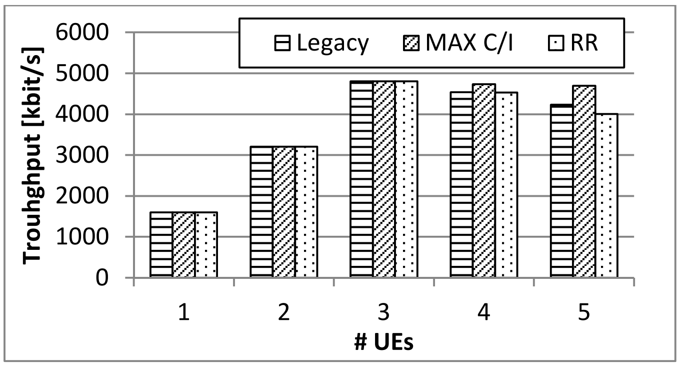

Fig. 10. Average cell throughput with three scheduling algorithms, 1 to 5 UEs, and CBR traffic with a payload size of 150 Bytes.

IV. System profiling

In this section we present the results of a profiling of the OAI system emulation platform. More specifically, we evaluate the execution time and the memory occupancy of **oaisim**. Our goal is to uncover potential performance bottlenecks and propose ameliorations.

We first evaluate the execution time and memory occupancy as a function of the number of UEs and their traffic profile. The emulation is performed on a machine with an AMD FX 8350 4 GHz CPU, 8 GB RAM, running Xubuntu 14.04.1. The main emulation parameters are summarized in TABLE IV. Each scenario is run three times with three different seeds, for a total of nine runs per configuration of the factors.

We use two different set of tools, trading detail for invasiveness. The first tool is the `/usr/bin/time` command [10], which is minimally invasive, but offers less detailed insight. The latter measures both the execution time, as the name suggests, *and* memory usage. More specifically, it reports the *maximum resident set size* (maxRSS), i.e. the maximum amount of memory that the process has allocated in RAM. This allows us to roughly estimate the memory usage of the process with no impact on its execution time. A more detailed profiling can instead be done using the `valgrind` framework [11]. More specifically we use the tool `callgrind` to obtain the percentage of the total CPU time spent by *each* C function. Finally we use `massif` to obtain a detailed monitoring of the heap usage over time. The two above operations are very invasive: the former slows down the execution up to 100 times, the latter up to 20 times.

Fig. 11 shows the execution time with an increasing number of UEs. As we can see the time increase almost linearly with the number of UEs in all the considered scenarios. However the main impact on performance is by far due to the PHY mode in use. The same phenomenon can be observed when considering memory occupancy, as shown in Fig. 12. The above results confirm the ones available in the literature and clearly demonstrate that lightweight PHY emulation is important. Moreover, according to [6], the PHY abstraction mode introduces only negligible errors, hence is accurate enough. For the reasons above, we will now focus on profiling the system when PHY abstraction is enabled. As a side note, Fig. 11 shows that the ratio between the emulated and the real time is below one (recall that we are emulating 20s) when one or two users are emulated on the same machine. This implies that it is viable to run an eNB using the OAI framework in real time using commodity hardware. In fact, the same code as for the **oaisim** eNB is used in OAI real-time mode, *without* the burden of UE emulation and channel emulation (the latter being real-life when using antennas).

TABLE IV. EMULATION PARAMETERS USED FOR BENCHMARK

Parameter	Value
Emulation duration	20000 TTIs
Duplexing mode	FDD
PHY abstraction	Yes, no
# eNBs	1
# UEs	1 to 6
Mobility	Static
Traffic type	null, full buffer, CBR (100 and 200-byte payload)
# runs	9

Published at "3rd International Conference on Future Internet of Things and Cloud", © 2015 IEEE DOI 10.1109/FiCloud.2015.77

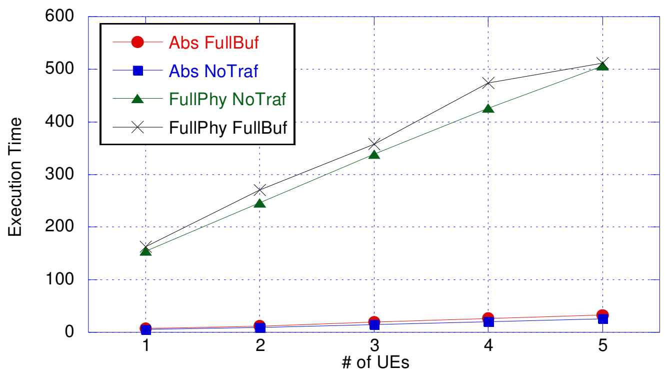

Fig. 11. Execution time with and without PHY abstraction mode, 1 to 5 UEs, with full buffer and no traffic configurations.

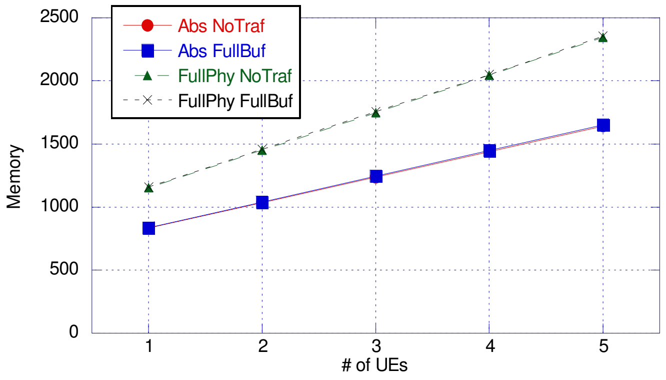

Fig. 12. Memory occupancy (maxRSS) with and without PHY abstraction mode, 1 to 5 UEs, with full buffer and no traffic configurations.

As we can see in Fig. 13, the execution time grows linearly with the number of UEs in case of PHY abstraction mode too: OAI in fact emulates one physical layer per active UE. Moreover the time increases when traffic generator are activated, with the *full-buffer* one placing higher burden than the CBR one. What is instead less intuitive is that the performance of the CBR traffic generator seems not to depend on the amount of traffic generated, i.e. it does not change when doubling the rate from 100 to 200 byte each ms. Fig. 14 shows the maximum memory occupation, which depends linearly on the number of active UEs, and not on the traffic.

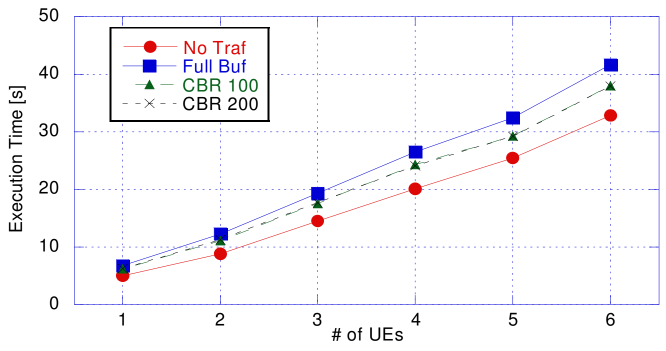

Fig. 13. Execution time with 1 to 6 UEs, 4 traffic configurations and PHY abstraction mode enabled.

Published at “3rd International Conference on Future Internet of Things and Cloud”, © 2015 IEEE DOI 10.1109/FiCloud.2015.77

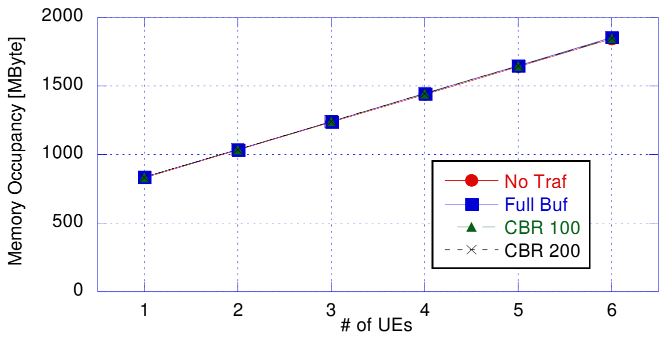

Fig. 14. Memory occupancy (maxRSS) with 1 to 6 UEs, 4 traffic configurations and PHY abstraction mode enabled.

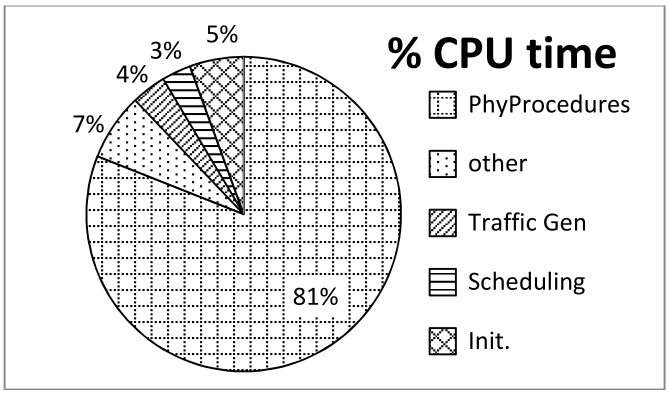

Fig. 15. Percentage of CPU time used by each OAI component during an emulation with 1 eNB, 1 UE, PHY abstraction mode enabled and CBR traffic.

```
v1*pow(x,7)+v2*pow(x,6)+v3*pow(x,5)+...+v6*pow(x,2)
```

```
x[]={ x , 1 , 1 , 1 , 1 , 1 , 1};
for(i=1 ; i < 7 ; ++i)
    x[i] = x[i-1]*x;
v1*x[6]+v2*x[5]+v3*x[4]+...+v6*x[1];
```

Fig. 16. Exemplification of the original (top) and optimized (bottom) versions of the code.

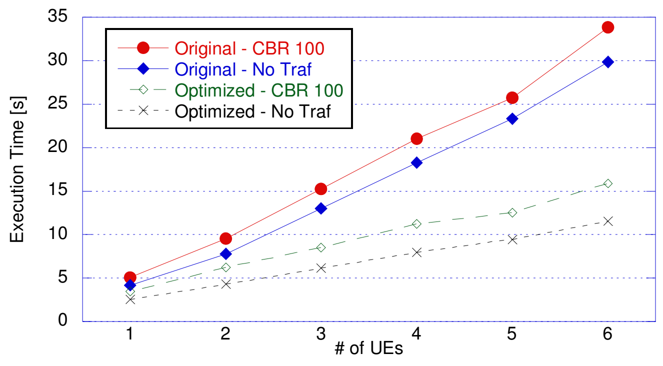

Fig. 17. Comparison between the execution time of the original OAI implementation against our optimization.

Published at “3rd International Conference on Future Internet of Things and Cloud”, © 2015 IEEE DOI 10.1109/FiCloud.2015.77

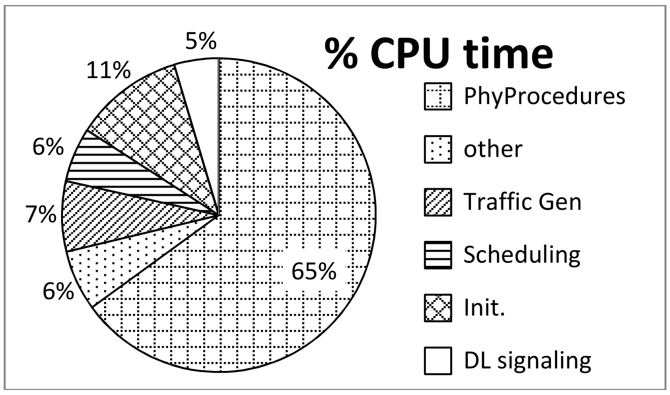

Fig. 18. Percentage of CPU time used by each OAI component, after code optimization, during an emulation with 1 eNB, 1 UE, PHY abstraction mode enabled and CBR traffic.

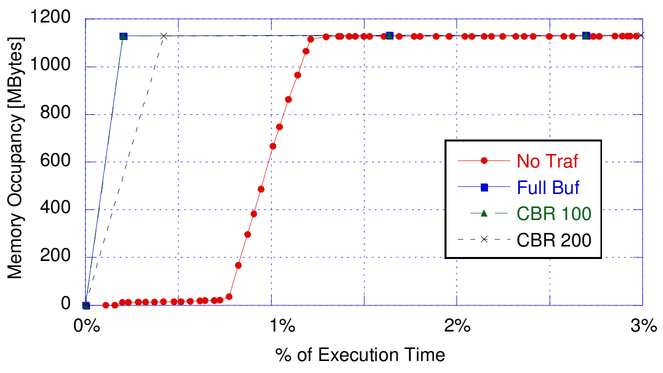

Fig. 19. Heap usage over time

In order to analyse the impact of each component of OAI on the execution time we use the tool `cachegrind`, and we monitor the execution of an emulation with 1 eNB, 1 UE, PHY abstraction mode enabled and CBR traffic with 100 bytes/ms. In Fig. 15 we show the percentage of CPU time used by the main OAI functions. First of all we observe that most of the execution time is spent in PHY-related operations, despite using lightweight PHY abstraction mode, confirming our previous results. If we look deeper into this, we discover that 67% of the total time is spent in function `sinr_eff_cqi_calc`, which computes the CQIs based on the SINR. Within the latter, 48% of the total time is spent in computing powers of double-precision real numbers, performed using the `pow()` function of glibc-2.19.

Some basic code optimization shows that the `pow()` function can be dispensed with most of the times. In fact, since *all* powers up to seven are required, one might as well use a cycle of multiplications, which is far more efficient. We show an exemplification of the original and optimized code in the upper and lower part of Fig. 16 respectively.

This halves the execution time without obviously affecting the results. Fig. 17 shows the execution time of the emulation before and after the code optimization. Fig. 18 shows instead the percentage of CPU time used by the main OAI functions *after* our code optimization. The modified proportions allow one to appreciate that the second-largest contribution is given by the initialization function, which eats up 11% of the total CPU time in an experiment with 20000 subframes. The reason for it becomes apparent by looking at Fig. 19, obtained using `massif`, which shows the heap memory allocated by OAI over time. As we can see, most of the memory is allocated at the very beginning of the emulation, thus justifying the amount of time spent in the initialization function.

The third-largest contribution to the execution time is given by the traffic generator, confirming the results we showed in Fig. 13. The rest of the time is mainly spent for scheduling operations and downlink signalling. Note that changing the scheduler does not result in appreciable differences in computation time. This is understandable, since the complexity of schedulers might be significant only with a very large number of UEs, a number which is hard to emulate on a single machine, mainly due to insufficient memory: according to the linear model of Fig. 12, the memory occupancy required to emulate 100 UEs is in the order of 20 Gbytes.

V. Conclusions and Future Work

This paper presented some aspects of the OAI framework, which are relevant when the latter is used as a testbed to prototype resource allocation algorithms. More specifically, we studied and documented the OAI scheduling framework,

Published at “3rd International Conference on Future Internet of Things and Cloud”, © 2015 IEEE DOI 10.1109/FiCloud.2015.77

and implemented and validated two well-known MAC schedulers as a proof of concept. Moreover, we independently profiled the framework as for memory and execution time. Our profiling highlights that most of the time is spent executing PHY procedures, even with the lightweight PHY emulation mode. We proposed a simple optimization that halves the execution time of emulation campaigns. This biases us to think that a careful code re-engineering and optimization, especially of the PHY part, could lead to substantial reductions in the emulation time. This kind of profiling, even if performed in the oaisim package (the simulation/emulation environment of OAI), is especially relevant in a future prototype implementation of a real-world testbed based on USRP hardware, with a different implementation of PHY layer, but where in any case upper layers are the same, and we would like to save as many resources as possible for advanced tasks (e.g., coordination algorithms).

Future work – to be carried out in the framework of the Flex5GWare EU H2020 project – include using OAI in a C-RAN environment, where coordinated instances of eNBs are setup dynamically in response to network events.

Acknowledgment

The subject matter of this paper includes description of results of a joint research project carried out by Telecom Italia and the University of Pisa. Telecom Italia reserves all proprietary rights in any process, procedure, algorithm, article of manufacture, or other result of said project herein described.

This work has been performed in the framework of the Flex5GWare EU H2020 project, which is partly founded by the European Union.

Appendix A

The pseudo code of the *legacy* scheduler implemented within OAI is given in the following snippet.

```
Let $n$ be the number of active UEs
for all $i \in \{0, n\}$
    Let $CQI_i$ be the $i$'s CQI
    Let $r_i$ be the max round for $i$'s active HARQ processes
    Let $SBR_i$ be $i$'s buffer status as reported in SRBs
    Let $t_i$ be $i$'s last serving time
    Let $B_i$ be the size of $i$'s buffer
    for all $j \in \{i+1, n\}$
        Let $CQI_j$, $r_j$, $SBR_j$, $t_j$, $B_j$
        if $r_i > r_j$
            swap($i$,$j$)
        else if $r_i = r_j$
            if $SBR_i < SBR_j$
                swap($i$,$j$)
            else if $t_i < t_j$
                swap($i$,$j$)
            else if $B_i < B_j$
                swap($i$,$j$)
            else if $CQI_i < CQI_j$
                swap($i$,$j$)
            end if
        end if
    end for
end for
```

References

[1] A. Checko, H.L. Christiansen, Ying Yan; L. Scolari, G. Kardaras, M.S. Berger, L. Dittmann, "Cloud RAN for Mobile Networks—A Technology Overview," Communications Surveys & Tutorials, IEEE , vol.17, no.1, pp.405,426, Firstquarter 2015

[2] “C-RAN: The Road Toward Green RAN”, China Mobile Research Institute (2011), Beijing, China, Oct. 2011, Tech Rep.

[3] Amarisoft. Amari LTE 100 - Software LTE base station on PC. url: http://www.amarisoft.com/?p=amarilte (accessed May 2015)

[4] A. Virdis, G. Stea, G. Nardini, "SimuLTE: A Modular System-level Simulator for LTE/LTE-A Networks based on OMNeT++", SimulTech 2014, Vienna, AT, August 28-30, 2014

[5] R. Wang *et al.* “OpenAirInterface - An effective emulation platform for LTE and LTE-Advanced”. In: Ubiquitous and Future Networks (ICUFN), 2014 Sixth International Conf. on. Shanghai, China: IEEE, 2014, pp. 127–132.

[6] I. Latif, F. Kaltenberger, N. Nikaein, R. Knopp. “Large scale system evaluations using PHY abstraction for LTE with OpenAirInterface”, in Proceedings of SimuTools '13. Brussels, Belgium, Belgium, 24-30. 2013.

[7] OpenAirInterface website, Url: http://www.openairinterface.org. (accessed May 2015)

[8] OpenAirInterface press release Url: http://www.openairinterface.org/openairfiles/openAirInterface_nov_2014.pdf (accessed May 2015)

[9] A. Hafsaoui, N. Nikaein, W. Lusheng, "OpenAirInterface Traffic Generator (OTG): A Realistic Traffic Generation Tool for Emerging Application Scenarios," Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on , vol., no., pp.492,494, 7-9 Aug. 2012

Published at “3rd International Conference on Future Internet of Things and Cloud”, © 2015 IEEE DOI 10.1109/FiCloud.2015.77

[10] M. Kerrisk., time(1) - Linux manual page. url: http://man7.org/linux/man-pages/man1/time.1.html. (accessed May 2015)

[11] Valgrind, url: http://valgrind.org/ (accessed May 2015)

[12] Massif, url: http://valgrind.org/docs/manual/ms-manual.html (accessed May 2015)

[13] Valgrind’s Tool Suite, url:http://valgrind.org/info/tools.html (accessed May 2015)

[14] Flex5Gware - Flexible and efficient hardware/software platforms for 5G network elements and devices, Funded by the EU under call H2020-ICT-2014-2.

[15] I. Alyafawi, E. Schiller, T. Braun, D. Dimitrova, A. Gomes, N. Nikaein, “Critical Issues of Centralized and Cloudified LTE-FDD Radio Access Networks”, ICC 2015, London, UK, June 8-12, 20