A Real-Time Vendor-Neutral Programmable Scheduler Architecture for Cellular Networks

Wenhao Zhang*, Zhouyou Gu*, Wibowo Hardjawana*, Branka Vucetic*,
Simon Lumb†, David McKechnie† and Todd Essery†
*School of Electrical and Information Engineering, The University of Sydney, Australia, †Telstra Corporation Ltd
Email: {wenhao.zhang, zhouyou.gu, wibowo.hardjawana, branka.vucetic}@sydney.edu.au
{simon.lumb, david.mckechnie, todd.essery}@team.telstra.com

***Abstract*—The current Downlink Shared Channel (DLSCH) resource scheduler for cellular networks has the following features: 1) it is integrated with an evolved NodeB (eNB) and 2) uses proprietary interfaces. The first causes a temporary outage whenever the scheduler logic is reprogrammed to accommodate traffic profiles that have different requirements, while the latter prevents multi-vendor interoperability. In this paper, we propose a real-time vendor-neutral programmable DLSCH scheduler architecture. The scheduler and eNB are separated into two binary files that communicate via an agent. The agent uses standard interfaces to interpret information from/to different eNB vendors in real time. The proposed architecture is implemented on two open source 3rd Generation Partnership Project standard-compliant eNB stacks from the OAI and SRS. Experimental results show that the proposed architecture addresses the real time and proprietary challenges mentioned above.**

***Index Terms*—Long Term Evolution, vendor-neutral, programmable scheduler, hardware implementation**

I. Introduction

Multiple use cases with time-varying downlink traffic profiles in the cellular networks, namely enhanced mobile broadBand (eMBB), massive machine type communications (mMTC) and ultra-reliable low-latency communications (URLLC), have different requirements in terms of packet size, priority, reliability and latency. They are carried by a transport channel referred to as the Downlink Shared Channel (DLSCH). The radio resources to transmit this channel are allocated by a scheduler every transmission time interval (TTI). The amount of allocated radio resources depends on the current traffic demands, which change from time to time. This implies that the DLSCH scheduler logic needs to be programmed so that the needed resources are matched with the traffic requirements.

Multiple DLSCH programmable schedulers that run at the edge base stations (BSs), referred to as evolved NodeBs (eNBs), have been developed in the past. They range from commercial cellular networks operated by commercial vendors to those built as part of experimental research networks [1]–[3]. These experimental networks are built on top of open source eNBs from the OpenAirInterface (OAI) [4], referred to as OAI eNB, and Software Radio Systems (SRS) [5], referred to as SRS eNB. All the above mentioned schedulers have three commonalities. First, the behaviour of these schedulers can only be changed from a controller of the network operators, either by selecting a pre-defined logic, such as round robin (RR) or proportional fair (PF) scheduling [1] or by changing pre-defined parameters such as priorities, minimum bitrates, packet delay budgets, etc. [2]. Second, they are tightly embedded into the eNBs, meaning that they run there as a single binary file. If the scheduler logic is re-programmed, the eNB needs to be recompiled; this leads to a temporary outage. Thus, these schedulers are *non-real-time programmable*. Third, the communication interface between the scheduler and the media access control (MAC) is proprietary; therefore, there is no multi-vendor interoperability. This means operators need to use schedulers from the same vendor as those used for eNBs, which is not desirable in commercial networks. Recently, the O-RAN Alliance proposed vendor-neutral interfaces for eNBs to allow multi-vendor interoperability between baseband processing and radio frequency (RF) units [6]. To the best of our knowledge, no attempt has so far been made to create a real-time vendor-neutral programmable DLSCH scheduler for cellular networks.

In this paper, we propose a Real-Time Vendor-Neutral (RTVN) programmable DLSCH scheduler architecture. The first contribution here is that the DLSCH scheduler logic at an eNB will become *real-time programmable*. This is achieved by separating the DLSCH scheduler from the eNB as a separate binary file, referred to as a shared library. The remaining eNB functions are then run in another binary file, referred to as ReNB. As a result of the separation, the logic of the DLSCH can be changed and compiled individually and in real time, without interrupting the operation of the eNB. This is in contrast with [1] and [2], where any change made to the scheduler logic will result in a temporary outage at the eNB. The second contribution is a vendor-neutral DLSCH scheduler that works with the ReNB vendor. This is achieved by standardising the interface parameters exchanged between the scheduler and ReNB, referred to as the global variable (GV). This is in contrast with [1] and [2], which employ proprietary interfaces. The third contribution is the development of a vendor-neutral controller that can generate the shared library containing the DLSCH scheduler logic. The scheduler logic is written by operators and complied as a shared library at the controller, using the GV as its interface parameters. The shared library is then sent to eNB and is used as a new scheduler. A prototype for the above architectures, running on different ReNBs from the OAI and SRS, is

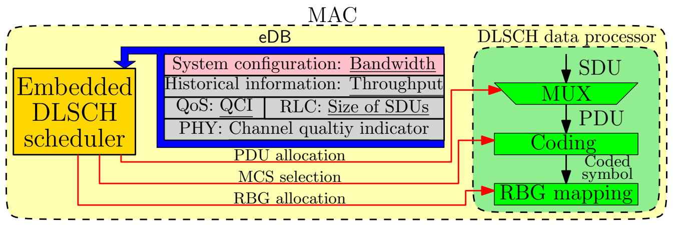

Fig. 1. eNB DLSCH scheduler and data processor

then developed. Corresponding agents are then developed inside these two ReNBs to 1) ensure the shared library can communicate with ReNB by using vendor-neutral interface parameters, GV; and 2) replace the shared library with a new one in real time. Several over-the-air Long Term Evolution (LTE) standard-compliant experiments using commercial user equipment (UE) are conducted to show the benefits of the proposed architecture over [1] and [2]. Note that the proposed architecture can be used to schedule other transport channels in LTE and in fifth generation (5G) networks by expanding the functionalities of the agent and the GV. The code suite of proposed vendor-neutral DLSCH scheduler and related interface is at https://github.com/harryzwh/ops_project.

The remainder of this paper is organised as follows. In Section II and Section III, we introduce the DLSCH scheduler for both the existing and the proposed RTVN eNBs. In Section IV, we describe the controller used for it. The prototype and its experimental results are presented in Section V. Section VI concludes the paper.

II. The DLSCH Scheduler and Data Processor

We will first explain the existing embedded eNB scheduler architecture. This architecture is shown in Fig. 1; it consists of an embedded DLSCH scheduler and its corresponding MAC layer functions (namely, eNB databases (eDB) and a DLSCH data processor). For simplicity, we follow the default configurations of open-source LTE experimental networks, the OAI eNB and SRS eNB: 1) a single-input-single-output (SISO) system (transmit mode 1); and 2) resource allocation type 0, which allocates radio resources to UEs based on the resource block group (RBG), consisting of multiple resource blocks (RBs).

The inputs used by the DLSCH scheduler are stored in eDB and grouped into two categories: the eNB system and the UE information. The eNB system stores information on available RBs in each TTI. For each instance of UE information, a historical downlink throughput, a channel quality indicator (CQI) and data radio bearer (DRB) information are stored. DRB information consists of the packet size of the Service Data Units (SDUs) and quality of service (QoS) Class Identifier (QCI) for each DRB. Note that SDUs are packets that carry the user payload in the radio link control (RLC) layer. Depending on the scheduler logic employed by eNB vendors, the DLSCH scheduler outputs are 1) modulation coding scheme (MCS) selection; 2) size of the protocol data unit (PDU) (PDU allocation); and 3) allocated RBGs (RBG allocation).

We now explain how the above scheduler outputs are used by the DLSCH data processor in every TTI. The DLSCH data processor consists of MUX, coding and RBG mapping functions. For each DRB, the MUX function multiplexes SDUs into a PDU where the total size of the SDUs is limited by the size of PDU, which is taken from PDU allocation. Coding function is used to convert bits in PDU from the same UE into coded symbols. The coding scheme is decided by MCS selection. Finally, the RBG mapping function maps coded symbols belonging to different UEs to allocated RBG based on the information provided by the RBG allocation. This data flow is shown in Fig. 1. Note that in both the OAI eNB and SRS eNB, all the above components run as a single binary file with proprietary interfaces. This means 1) the eNB will need to be recompiled every time if the objective of the DLSCH logic is changed, resulting in an operational outage; and 2) a non-vendor-neutral architecture where the vendor of scheduler must be the same with the one of the eNB.

III. The RTVN DLSCH Scheduler Architecture

The RTVN DLSCH scheduler architecture consists of a vendor-neutral eNB and controller. In this section, we will explain the vendor-neutral eNB and leave the explanation about the controller to Section IV. There are three components in the eNB: 1) a separate DLSCH scheduler to ensure the real-time operation of ReNB; 2) a global variable that consists of standardised input and output parameters to guarantee vendor neutrality; and 3) an agent to manage the above two components. The proposed architecture is shown in Fig. 2.

A. DLSCH Scheduler Separation

The DLSCH scheduler separation is achieved by moving the embedded DLSCH scheduler out from the MAC in Fig. 1 as a shared library. The remaining eNB functions inclusive of MAC layer functions (i.e., the eDB and DLSCH data processor) are defined as ReNB. The shared library and ReNB are linked by using standardised input and output parameters. These parameters are abstracted and interpreted using an agent that has been specifically developed for different ReNB vendors. By so doing, the scheduler and ReNB can now be compiled separately as two binary files and the scheduler can be replaced without recompiling and interrupting ReNB.

B. Global Variables

We will now explain how to standardise the input and output parameters above, collectively defined as GV, for ReNBs from different vendors. GV stores system bandwidth, bw, represented by the number of RBGs in a subframe. For UE$_i$, $i = 1, \cdots, n$, GV contains $rnti_i$, cqi_i and tp_i as inputs, where n is the number of UEs in the system. $rnti_i$ is the radio network temporary identifier (RNTI). cqi_i is the channel information that follows the standard wideband CQI defined in the Third-Generation Partnership Project (3GPP) [7]. tp_i is the historical downlink throughput defined as the size of the

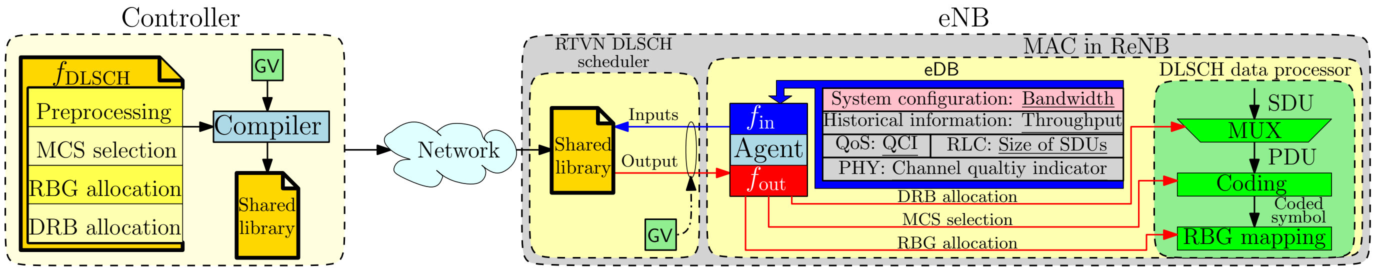

Fig. 2. RTVN programmable scheduler architecture

TABLE I
DLSCH SCHEDULER PARAMETERS IN GV

Name	Description
Input list	
$rnti_i$	The RNTI of UE$_i$
cqi_i	The CQI of UE$_i$
sdu_i^j	Size of SDUs of DRB$_i^j$ in byte in RLC buffer
qci_i^j	The QCI of DRB$_i^j$
tp_i	The throughput of UE$_i$ in byte/TTI
bw	The system bandwidth in the number of RBGs
Output list	
x_i	The allocated RBG bit-mask of UE$_i$
mcs_i	The MCS of UE$_i$
pdu_i^j	Size of PDU in byte DRB$_i^j$ can transmit

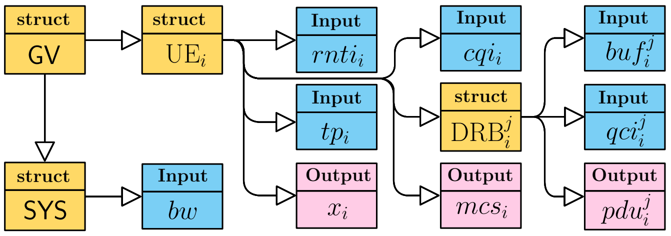

Fig. 3. Structure of GV

PDUs measured by the number of bytes that are sent from the RLC layer to the MAC layer in the previous TTI. For each DRB^j, $j = 1, \cdots, p$, of UE$_i$, DRB$_i^j$, GV has two inputs, sdu_i^j and qci_i^j. p is the maximum number of DRB defined in 3GPP [8]. sdu_i^j is the size of SDUs in number of byte that queued in the RLC layer downlink buffer. qci_i^j is the QCI assigned to DRB$_i^j$, which follows the definition in the 3GPP. All above input parameters are summarised in the input list in Table I. For outputs, mcs_i is the MCS of UE$_i$ that is decided by the scheduler and x_i is the bit-mask of RBG allocation for UE$_i$, indicating the positions of the allocated RBGs. pdu_i^j is defined as the size of the PDU that DRB$_i^j$ can transmit. The list of output parameters are summarised in the output list in Table I. We organise all parameters in Table I as a hierarchical data structure, shown in Fig. 3.

C. Agent

To manage the GV, we develop agents for different ReNB vendors. The agents have two functions, f_{in} and f_{out}. To send the scheduling inputs to the scheduler, f_{in} abstracts information from vendor-specific eDB and formats this information according to the GV, as shown in the input list in Table I. The implementation of f_{in} is shown in Algorithm 1. All input information are collected from the eDB and the output is GV. f_{in} first initialises and configures the data structure of GV and SYS. f_{in} then gathers the information of UEs and the DRB of each UE from the eDB in lines 4–13. UE$_i$ will be set as an empty *struct* if it does not have downlink data to transmit in the current TTI.

Algorithm 1: The logic of f_{in}

```
Input: eDB
Output: GV
1  Initialise GV = ∅, SYS = ∅;
2  Get $bw$ from eDB and set it to SYS;
3  Attach SYS to GV;
4  for $i = 1$ to $n$ do
5      Initialise UE$_i$ = ∅;
6      if $\exists sdu_i^j > 0$ then
7          Get $rnti_i$, $cqi_i$, $tp_i$ from eDB and set them to UE$_i$;
8          for $j = 1$ to $p$ do
9              Initialise DRB$_i^j$ = ∅;
10             if $sdu_i^j > 0$ then
11                 Get $sdu_i^j$, $qci_i^j$ from eDB and set them to UE$_i$;
12                 Attach DRB$_i^j$ to UE$_i$;
13         Attach UE$_i$ to GV;
14 return GV;
```

To control the DLSCH data processors from different vendors, f_{out} interprets x_i, mcs_i and pdu_i^j , shown in the output list in Table I, into RBG allocation, MCS selection and PDU allocation that the DLSCH data processors of ReNB vendors can understand. The implementation of f_{out} is shown in Algorithm 2. With vendor-specific f_{in} and f_{out} run in the agent to handle the above processes, the RTVN DLSCH scheduler can now communicate with any vendor's ReNB. Note that GV, f_{in} and f_{out} in this paper cover the parameters used by the DLSCH scheduler in LTE eNB only. They can be extended to support the schedulers of other transport channels and cell configurations (e.g. multi-input multi-output (MIMO)) and 5G BSs by adding more parameters in GV and expanding the functionalities of f_{in} and f_{out}.

Algorithm 2: The logic of f_{out}

```
Input: GV
Output: RBG allocation, MCS selection and PDU allocation
1 foreach $\text{UE}_i$ do
2     if $\text{UE}_i \neq \varnothing$ then
3         Get $x_i$ from GV to generate RBG allocation;
4         Get $mcs_i$ from GV to generate MCS selection;
5         for $j = 1$ to $p$ do
6             if $\text{DRB}_i^j \neq \varnothing$ then
7                 Get $pdu_i^j$ from GV to generate PDU allocation;
8 return RBG allocation, MCS selection and PDU allocation;
```

IV. RTVN DLSCH Scheduler Controller

Using examples, we now explain how the DLSCH scheduler logic with GV serving as its interface parameters is written at the controller for different ReNB vendors. The logic is compiled as a shared library and sent to replace the existing one at ReNB. It interacts with the eDB and DLSCH data processor via GV using f_{in} and f_{out}. The scheduler logic consists of four components. Pre-processing is done first in order to calculate the number of bytes of SDUs to be transmitted and the average throughput. It also initialises the RBG allocation bit-mask. mcs_i, x_i and pdu_i^j are then calculated for MCS selection, RBG allocation and PDU allocation, respectively. The above process is shown in Fig. 2. We refer to this scheduling function as f_{DLSCH}. f_{DLSCH} can be customised according to the operator need, making the scheduler logic programmable.

A. Pre-processing

In each TTI, the amount of transmitted data of UE_i, $queue_i$, is calculated by f_{mux} with sdu_i^j, $j = 1, \cdots, p$ as inputs.

$$queue_i = f_{\text{mux}}\left(sdu_i^1, \cdots, sdu_i^p\right) = \sum_{j=1}^{p} sdu_i^j \ . \qquad (1)$$

The average throughput of UE_i, avg_i, is calculated based on avg_i in the previous TTI, and the historical throughput, tp_i, defined in the input list in Table I. avg_i is set to 0 at the beginning,

$$avg_i = f_{\text{avg}}\left(avg_i, tp_i\right) = (1-\alpha)\, avg_i + \alpha \times tp_i \qquad (2)$$

where $\alpha = 0.99$. Note that the bit-mask of UE_i is set to 0, $x_i = 0$, to make sure that no RBG is allocated to any UE before RBG allocation takes place.

B. MCS Selection

After pre-processing, mcs_i is calculated for each UE_i using f_{mcs} with cqi_i, defined in the input list in Table I, as an input in lines 4–5 of Algorithm 3. In this paper, we adopt the

Algorithm 3: The logic of f_{DLSCH}

```
Input: GV
Output: GV
/* Preprocessing */
1 for $i = 1$ to $n$ do
2     $queue_i = f_{\text{mux}}\left(sdu_i^1, \cdots, sdu_i^p\right)$;
3     $avg_i = f_{\text{avg}}\left(avg_i, tp_i\right)$;
4     $x_i = 0$;
/* For MCS selection */
5 for $i = 1$ to $n$ do
6     $mcs_i = f_{\text{mcs}}\left(cqi_i\right)$
/* For RBG allocation */
7 for $k = 1$ to $bw$ do
8     for $i = 1$ to $n$ do
9         $m_i = f_{\text{m}}\left(bw, mcs_i, avg_i, x_i, qci_i^1, \cdots, qci_i^2, queue_i\right)$;
10    $i' = f_{\text{a}}\left(m_1, \cdots, m_n\right)$;
11    $x_{i'} = x_{i'} + 2^{k-1}$;
/* For PDU allocation */
12 for $i = 1$ to $n$ do
13    for $j = 1$ to $m$ do
14        $pdu_i^j = f_{\text{q}}\left(bw, mcs_i, x_i, qci_i^j, queue_i\right)$;
15        Update $pdu_i^j$ to GV;
16    Update $x_i$ to GV;
17    Update $mcs_i$ to GV;
18 return GV;
```

mapping function between CQI and MCS used by the OAI eNB in [9].

C. RBG Allocation

RBG allocation takes place in lines 7–11 of Algorithm 3 to calculate x_i. It consists of two functions: f_{m}, which calculates a utility value, m_i, of UE_i and f_{a} that selects UEs to which to allocate RBGs based on calculated utility values. f_{m} and f_{a} are run iteratively to select UEs and allocate RBGs to chosen UEs, one at a time. We will provide two examples, referred to as PF and traffic prioritisation (TP), to explain the RBG allocation process. For the first example, we consider a PF scheduling where the utility value is defined as the ratio between instantaneous throughput and average throughput. The RBG resource will be allocated to the UE with the highest utility value in each iteration. In the iteration k, utility values, m_i, $i = 1, \cdots, n$, are calculated by function

$$\begin{aligned} m_i &= f_{\text{m}}^{\text{PF}}\left(bw, mcs_i, x_i, avg_i, queue_i\right) \\ &= \begin{cases} \frac{f_{\text{TBS}}(bw, mcs_i, x_i)}{avg_i}, & \text{if } f_{\text{TBS}}\left(bw, mcs_i, x_i\right) < queue_i \\ -\infty, & \text{otherwise.} \end{cases} \end{aligned} \qquad (3)$$

Here, avg_i is the average throughput and $f_{\text{TBS}}\left(bw, mcs_i, x_i\right)$ is used to calculate the instantaneous throughput. The instantaneous throughput is represented by the available transport block size (TBS) in the current TTI based on the allocated RBGs, x_i, the number of RB per RBG determined by bw and the MCS, mcs_i, defined in Table I. The calculation is achieved by following the 3GPP standard (Table 7.1.6.1-1, Table 7.1.7.1-1 and Table 7.1.7.2.1-1 in [10]. We then use f_{a} to find the index, i, of the UE who has the largest utility value. f_{a} is defined as

$$i' = f_{\mathrm{a}}\left(m_1, \cdots, m_n\right) = \arg\max\left(m_1, \cdots, m_n\right) \quad (4)$$

For m_i with the same value, the one that has the smaller index will be selected. RBG k is then allocated to UE$_i$ by setting the bit k of x_i to 1. As the RBG index, k, is increased by one in each iteration, each RBG will only be allocated to one UE. For the second example, we consider a TP scheduling where RBGs will be allocated to the UE with the highest priority. We use the QCI of the first DRB, qci_i^1, to represent the priority of UE$_i$. The utility values, m_i, $i = 1, \cdots, n$, are calculated as

$$\begin{aligned} m_i &= f_{\mathrm{m}}^{\mathrm{TP}}\left(bw, mcs_i, x_i, qci_i^1, queue_i\right) \\ &= \begin{cases} -qci_i^1, & \text{if } f_{\mathrm{TBS}}\left(bw, mcs_i, x_i\right) < queue_i \\ -\infty, & \text{otherwise.} \end{cases} \end{aligned} \quad (5)$$

(4) then is used to find the selected UE, UE$_i$. In this case, the UE with the highest priority (the smallest QCI) will always be selected, while lower-priority UEs with will be blocked.

D. DRB Allocation

DRB allocation is achieved using f_{q}, which computes pdu_i^j in lines 12–17 of Algorithm 3 to indicate how many bytes DRB$_i^j$ can transmit in a PDU. We provide an example of f_{q} for a single DRB scenario. In this example, all RBG resources are allocated to the first DRB of UE$_i$, DRB$_i^1$, so that DRB$_i^1$ can transmit a PDU with a size equal to TBS. f_{q} takes bw, defined in the input list in Table I, mcs_i, calculated during MCS selection, and x_i, calculated during RBG allocation as inputs to calculate the TBS of UEi. f_{q} is defined as

$$pdu_i^1 = f_{\mathrm{q}}\left(bw, mcs_i, x_i\right) = f_{\mathrm{TBS}}\left(bw, mcs_i, x_i\right) . \quad (6)$$

All four components of f_{DLSCH} presented in Sections IV-A–IV-D are summarised in Algorithm 3.

E. The Shared Library

The shared library is written as a customised f_{DLSCH} based on the GV structure at the controller by network operators. The source code of the shared library is then compiled as a binary file. The compiler will take different CPU architectures (e.g., X86 and ARM) into consideration and generate different shared libraries accordingly from the same source code. This enables target neutrality, meaning that the shared library can work with multiple ReNBs built on different CPU architectures. The shared library is sent to ReNB and used by the agent there to replace the old in real time.

V. Prototype

We now describe the prototype development of the RTVN programmable scheduler architecture. We create two ReNBs by modifying the OAI eNB and SRS eNB. They are run on dedicated computers, located in our lab. Both computers connect to a Universal Software Radio Peripheral (USRP) B210 via USB 3.0 at the RF end. The Evolved Packet Core (EPC) (using an open source EPC from [11]) is operated on another computer in the lab and connected to the same internal network as the ReNBs via the Ethernet. The controller also runs on a computer in the internal network. It is used to provide a web-based interface for writing the RTVN DLSCH scheduler in C and compiling the source code into a shared library. The compiled shared library is then distributed to ReNBs using a secure copy (SCP) protocol, where SCP is an application layer protocol used to copy files between computers. We use commercial phones as UEs in our experimental network. The experimental network runs in LTE Band 7, with an operating bandwidth of 10 MHz (i.e., bw = 17). Each UE is configured to run an application that tries to maximise the downlink throughput. The experimental LTE network deployed in our lab is shown in Fig. 4

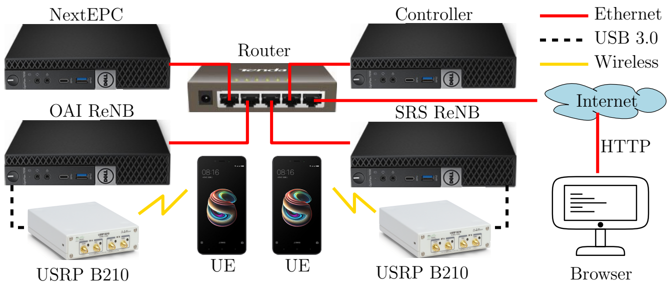

Fig. 4. Experimental setup of LTE network

A. The RTVN DLSCH Scheduler for Different ReNB Vendors

The first experiment we run demonstrates vendor neutrality. In this experiment, we consider vendor neutrality to have been achieved when the OAI and SRS ReNBs have the same downlink throughput when the same scheduler is used. We assume there are two UEs, attached to each ReNB. Each UE uses one DRB and is configured with different priorities. For each ReNB, UE$_1$ has $qci_1^1 = 8$ as a high-priority UE and UE$_2$ has $qci_2^1 = 9$ as a low-priority UE. Two UEs are placed in different locations so that UE$_1$ has worse channel quality with measured CQI, $cqi_1 = 10$, and UE$_2$ has better channel quality with measured CQI, $cqi_2 = 15$. The experimental results are shown in Fig. 5. In the first 5000 TTIs, we run PF scheduling using (3) and (4) as f_{m} and f_{a} in f_{DLSCH}. Both ReNBs provide UE$_2$ with a higher throughput due to the higher CQI, which results in better spectrum efficiency and more allocated resources. The scheduling function update is triggered to run TP at TTI=5000 by using (5) and (4) as f_{m} and f_{a} in f_{DLSCH}. In that case, for both ReNBs only UE$_1$ has downlink traffic due to a higher priority. The throughput difference between OAI ReNB and SRS ReNB is due to the different control format indicator (CFI) configurations. In SRS ReNB, CFI is fixed at 3, which means the first three orthogonal frequency-division multiplexing (OFDM) symbols in each subframe are used for the control channel, regardless of the amount of downlink control information (DCI). While in OAI ReNB, CFI configuration is dynamic depending on the traffic demand. With only two UEs, a smaller CFI configuration

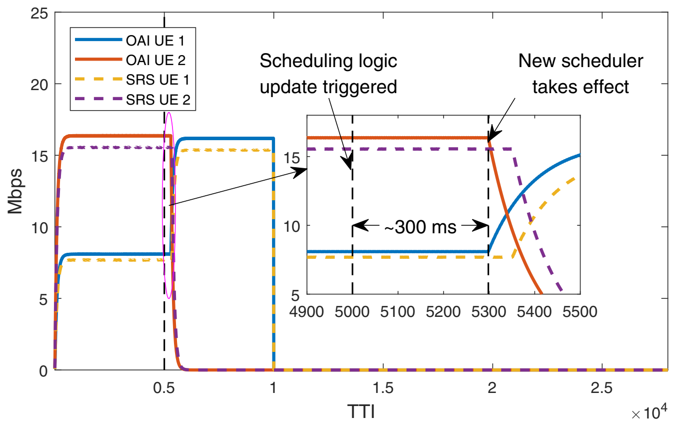

Fig. 5. Throughput of different schedulers in RTVN architecture

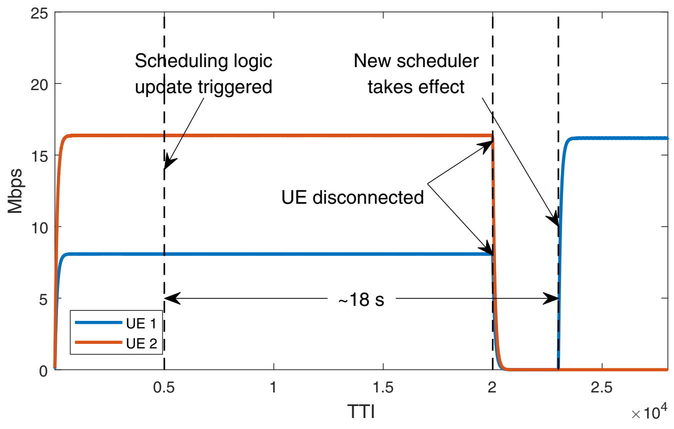

Fig. 6. Throughput of different schedulers in FlexRAN [1]

value is sufficient to carry all DCI; thus, more resources can be used to transmit the UE payload.

B. Real-Time Scheduling

The second experiment demonstrates real-time programmability by comparing the scheduling latency between our RTVN architecture to a state-of-the-art experimental embedded scheduler architecture, FlexRAN [1]. The latency is defined as the time taken between triggering a new logic to running the new logic at the eNB. The same setup as used in Section V-A is used for FlexRAN. The experimental results of FlexRAN are shown in Fig. 6. FlexRAN takes about 18s to recompile and re-starting the eNB after the new logic is triggered at TTI=5000. During the re-start process, the operation of eNB is terminated so that the UEs are disconnected and do not receive any traffic. For RTVN, as shown in Fig. 5 before the new scheduler is run the original PF continues to run and maintains the UE connections. The latency is about 300 ms for the shared library's compilation and distribution. Note that, for SRS ReNB, the latency is slightly longer because we send the shared library to the ReNBs one by one. This latency can be further reduced by optimising the compilation process and using faster transmission protocols.

VI. Conclusion

In this paper, we address the challenge of programming the scheduler in multi-vendor cellular networks by proposing a real-time vendor-neutral programmable architecture and implementing it for both the OAI eNB and SRS eNB. The proposed architecture separates the embedded DLSCH scheduler from eNB and eliminates proprietary interfaces. This allows the RTVN DLSCH scheduler to be programmed in a vendor-neutral manner and in real time at the controller. The experimental prototyping activities have shown that the proposed architecture provides a high degree of programmability in scheduler logic and allows operators to apply customised strategies according to their needs in real time.

Acknowledgement

This project, titled ”Development of an Open Programmable Scheduler for LTE Networks” is directly funded by Telstra Corporation Ltd. and supported by an Australian Research Council Discovery Early Career Research Award (DE150101704). It is also partially supported by an Australian Research Council Laureate Fellowship grant number FL160100032. The support provided by University of Sydney International Scholarship (USydIS) of Wenhao Zhang is acknowledged.

References

[1] X. Foukas, N. Nikaein, M. M. Kassem, M. K. Marina, and K. Kontovasilis, “FlexRAN: A flexible and programmable platform for software-defined radio access networks,” in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 2016, pp. 427–441.

[2] F. Kaltenberger, C. Roux, M. Buczkowski, and M. Wewior, “The openairinterface application programming interface for schedulers using carrier aggregation,” in *2016 International Symposium on Wireless Communication Systems (ISWCS)*. IEEE, 2016, pp. 497–500.

[3] L. Gavrilovska, V. Rakovic, and D. Denkovski, “Aspects of resource scaling in 5g-mec: Technologies and opportunities,” in *2018 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2018, pp. 1–6.

[4] N. Nikaein, M. K. Marina, S. Manickam, A. Dawson, R. Knopp, and C. Bonnet, “OpenAirInterface: A flexible platform for 5G research,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 33–38, 2014.

[5] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith, “srsLTE: an open-source platform for LTE evolution and experimentation,” in *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*. ACM, 2016, pp. 25–32.

[6] O-RAN Alliance , “O-RAN Architecture,” 2018. [Online]. Available: https://www.o-ran.org/

[7] Evolved Universal Terrestrial Radio Access (E-UTRA), “Policy and charging control architecture (3GPP TS 23.203 version 8.9.0 Release 8),” *ETSI, Standard*, vol. 8, no. 9, 2010.

[8] ——, “Radio resource control (RRC) (3GPP TS 36.331 version 8.6.0 Release 8),” *ETSI, Standard*, vol. 8, no. 6, 2009.

[9] OpenAirInterface, 2018. [Online]. Available: https://gitlab.eurecom.fr/oai/openairinterface5g/blob/master/openair1/PHY/phy_vars.h

[10] Evolved Universal Terrestrial Radio Access (E-UTRA), “Physical layer procedures (3GPP TS 36.213 version 8.6.0 Release 8),” *ETSI, Standard*, vol. 8, no. 6, 2009.

[11] S. Lee, J. Park, and J. B. Lee, “NextEPC,” 2018. [Online]. Available: http://nextepc.org/