RT-OPEX: Flexible Scheduling for Cloud-RAN Processing

Krishna C. Garikipati Kassem Fawaz Kang G. Shin

University of Michigan

ABSTRACT

It is cost-effective to process wireless frames on general-purpose processors (GPPs) in place of dedicated hardware. Wireless operators are decoupling signal processing from basestations and implementing it in a cloud of compute resources, also known as a *cloud-RAN* (C-RAN). A C-RAN must meet the deadlines of processing wireless frames; for example, $3ms$ to transport, decode and respond to an LTE uplink frame. The design of baseband processing on these platforms is thus a major challenge for which various processing and real-time scheduling techniques have been proposed.

In this paper, we implement a medium-scale C-RAN-type platform and conduct an in-depth analysis of its real-time performance. We find that the commonly used (e.g., partitioned) scheduling techniques for wireless frame processing are inefficient as they either over-provision resources or suffer from deadline misses. This inefficiency stems from the large variations in processing times due to fluctuations in wireless traffic. We present a new framework called RT-OPEX, that leverages these variations and proposes a flexible approach for scheduling. RT-OPEX dynamically migrates parallelizable tasks to idle compute resources at runtime, reducing processing times and hence deadline misses at no additional cost. We implement and evaluate RT-OPEX on a commodity GPP platform using realistic cellular workload traces. Our results show that RT-OPEX achieves an order-of-magnitude improvement over existing C-RAN schedulers in meeting frame processing deadlines.

Keywords

Cellular networks; Cloud-RAN; Real-time scheduling

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA

© 2016 ACM. ISBN 978-1-4503-4292-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2999572.2999591

1. INTRODUCTION

The baseband architecture of today's wireless networks is highly inefficient. Basestations use hardware that is usually proprietary, expensive, and difficult to upgrade. More importantly, the hardware resources (such as CPUs, DSPs, etc.) at each basestation are provisioned for their peak usage. This often results in severe resource-underutilization, as wireless traffic is known to exhibit significant spatial and temporal fluctuations within a network [19]. Moreover, as network density increases with deployment of small cells, the operating costs for the maintenance of the hardware (cooling, site visits, etc.) increases rapidly. Consequently, wireless operators are decoupling baseband processing from basestations and implementing it in a centralized pool of compute resources. The idea is to perform radio access network (RAN) functions in a cloud or a datacenter, where resources can be managed more efficiently. This approach, also known as *Cloud-RAN* (C-RAN), has received considerable attention from the industry as a way to reduce network costs [2, 19].

C-RAN attributes most of its advantages to resource pooling in which the aggregate workload of a group of basestations is processed collectively. Resource pooling has been shown to achieve 22% reduction in compute resources [15]. However, the biggest challenge in a C-RAN comes from the timing constraints of frame processing. For example, in LTE, a basestation must process a received subframe within a hard deadline of $3ms$ before sending an acknowledgment.

Meeting deadlines is inherently tied to the design of the wireless frame processing, particularly the scheduling of frames on available processors and the degree of parallel processing. A scheduler must handle wireless frames that arrive periodically at a fixed rate (every $1ms$ in LTE) by assigning each frame to a computing resource, where each frame has a hard processing deadline. On the other hand, parallelism is another design dimension that enables real-time frame processing. A typical baseband chain consists of signal processing blocks, such as FFT, equalizer, and decoder; each of these blocks can be broken down into independent (sub)tasks that can execute concurrently. For instance, FFT operations can run in parallel across different antennas or symbols while well-known parallel algorithms can be applied to Turbo decoding [24].

State-of-the-Art. Existing solutions utilize different scheduling schemes to meet the C-RAN's timing constraints; these fall under *partitioned* scheduling [15, 31, 36]. Partitioned schedulers employ an upper bound on the frames' processing time (a.k.a. the worst-case execution time (WCET)) as the fixed processing time, which enables the design of optimal scheduling of basestations on multiple processors — a problem known to be NP-complete [11–13, 18, 22, 26, 28].

On the other hand, one could also use a *global* scheduler which maintains a shared queue and assigns each incoming frame to the next available processor according to a priority mechanism, such as first-in-first-out (FIFO) or earliest-deadline-first (EDF). Global schedulers are flexible in that they adapt to the available number of cores and the processing time variations [11, 22, 28].

Other existing systems [32,35] exploit parallelism by splitting the baseband processing into parallel subtasks that can be executed on a large number of CPU cores. By achieving fined-grained parallelism and thus reduced processing times, scheduling incoming frames becomes straightforward as an incoming frame can finish processing before the next frame arrives.

Shortcomings. The assumption of fixed processing time, albeit being simple, does not hold in practice. Both basestation traffic and wireless channel state exhibit significant temporal and spatial variations that result in varying subframe processing times. For example, Fig. 1 shows the workload variations of Band-13 and Band-17 LTE basestations (in downlink) measured over a $50ms$ interval in a metropolitan region. As shown in the figure, the workload varies considerably between two consecutive subframes of a basestation, that are transmitted every $1ms$, in addition to variations across the two basestations. Therefore, a partitioned scheduler that relies on WCET (corresponding to peak load), will over-provision compute resources [27]. A global scheduler avoids the pitfalls of the partitioned scheduler by adapting to the variable processing time and available compute resources. Nevertheless, such schedulers are known to suffer from high runtime overhead (due to cache thrashing) [14].

Problem Statement. The compute resources in a typical C-RAN are made up of interconnected physical or virtual machines running on general-purpose processors (GPPs) and optionally, dedicated hardware accelerators. Similar to existing studies on C-RAN scheduling [15], we adopt the separation principle and decouple the problem of assigning basestations to computing nodes of a C-RAN from the problem of scheduling a subset of basestations on a single computing node. In this paper, we focus on the latter problem of scheduling tasks on a multiprocessor host, where each task represents a subframe decoding process of a basestation. In a C-RAN, the deadline misses result from the compute node not being able to decode a subframe within a deadline. In particular, we address the problem of reducing deadline misses at the node level by improving the performance of C-RAN schedulers.

Proposed Approach. To address the shortcomings of existing schedulers, we propose RT-OPEX (*Real-Time OPportunistic EXecution*), a new framework that combines offline scheduling with runtime parallelism. It minimizes deadline-misses of wireless frame processing by utilizing free CPU cycles at runtime to migrate parallelizable tasks to idle cores. The design of RT-OPEX is based on the premise that partitioned scheduling is unable to exploit free CPU cycles, while designing an offline scheduling algorithm with parallel processing is highly intractable. RT-OPEX treads the middle path and adapts offline partitioned scheduling at runtime to utilize the idle CPU cycles.

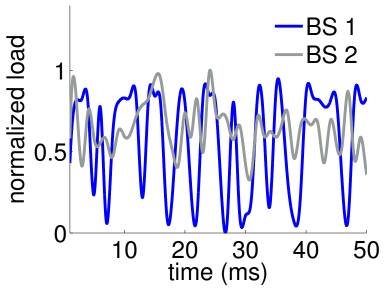


Fig. 1: Variations in cellular load traces.

RT-OPEX vs. Resource Pooling. Existing C-RAN literature [15, 19] has suggested resource pooling in which the statistical information of basestation loads is utilized to aggregate processing. We highlight that RT-OPEX is another variant of resource pooling, except that it consolidates processing at much finer time scales. Specifically, it utilizes the load variations of the order of subframes ($1ms$) to migrate processing on the compute platform. As a result, it maximizes the utilization of the available resources. However, unlike resource pooling, it has the advantage of making no assumptions about the prior knowledge of the load and traffic variations.

We evaluate the performance of RT-OPEX with other well-known schedulers: partitioned and global. For accurate evaluation, we implement a medium-scale testbed comprising 16 radios (frontends), off-the-shelf GPP platform, and Ethernet infrastructure. We profile our implementation to develop an end-to-end (e2e) model for processing that includes transport and processing latency of a wireless frame. Further, we evaluate the merits and demerits of each scheduling scheme under different system configurations, traffic loads, and channel conditions. Our implementation is available as an open-source package [9] that allows comparison of the performance of different C-RAN schedulers. Our results show that RT-OPEX, compared to existing partitioned and global scheduling schemes, reduces the deadline-miss rate by more than orders-of-magnitude.

Contributions. This paper makes the following contributions:

- development of an e2e model for characterizing the wireless frame processing times and the deadline-miss event;
- design, implementation, and evaluation of a flexible scheduling algorithm, RT-OPEX, which reduces deadline misses; and
- comparison of RT-OPEX with different schedulers for a medium-scale testbed under realistic workloads and scenarios.

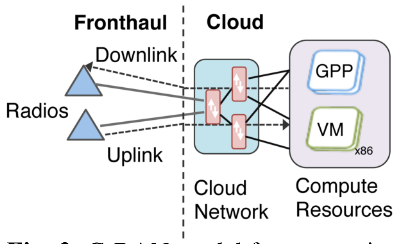


Fig. 2: C-RAN model for processing.

Organization of the paper. The rest of the paper is organized as follows. We present an end-to-end model of C-RAN, including frame processing and transport in Section 2. Section 3 describes the design and implementation of possible schedulers including our proposed RT-OPEX scheduler. Section 4 presents the evaluation results of our implementation. Section 5 discusses the features of the implemented schedulers. Finally, we discuss the related work in Section 6 and conclude the paper in Section 7.

2. END-TO-END MODEL

We assume a pure software-based C-RAN where the entire baseband processing (or L1) is carried out on general-purpose processors (GPPs) or virtual machines (VMs). Fig. 2 shows the main elements in a C-RAN deployment where baseband (or IQ) samples from the radios are transported back and forth over a fronthaul network. We first present a model to calculate the uplink processing time that allows us to characterize the deadline-miss event from an end-to-end perspective. This model is then used to develop C-RAN scheduling algorithms. We restrict our attention to uplink processing as it is significantly more time-consuming and varying than downlink [15, 25].

2.1 Uplink processing

A basestation, in the uplink, processes wireless signals from multiple antennas to decode user information. The basic unit of processing in LTE is a subframe ($1ms$ long). Each subframe can be viewed as a precoded sequence of 14 OFDM (Orthogonal Frequency-Division Multiplexing) symbols, which is divided into multiple physical resource blocks (PRBs). The PRBs are then assigned to one or more users; each user encodes information using a modulation and coding scheme (MCS) and transmits it in the allocated PRBs.

The computational load, and therefore, the time to process a subframe, is determined by the number of users, number of antennas, allocation of PRBs to users, MCS assignment, and the number of decoder iterations required to decode user information. To capture this relationship, we present a linear model that accurately approximates the processing time. We first establish the mapping between the MCS and the number of data bits.

Let us consider the transmission of a single user and let the subcarrier load, D, denote the ratio of the number of data bits (packet size) in a subframe to the number of resource elements (REs) available in a subframe, where RE is the basic data carrier unit in an LTE subframe. The packet size as a function of number of PRBs and MCS is determined by a lookup table specified in the LTE standard [7]. For 10MHz bandwidth, which has 8400 REs, D varies from 0.16 to 3.7 bits per RE (for 50 PRBs), corresponding to MCS 0 and MCS 27, respectively. The theoretical subcarrier load is 6 bits per RE when using 64-QAM modulation, but the realized load is much lower due to the overhead of coding, pilots and CRC bits.

LTE's uplink chain consists of commonly used signal processing blocks. To calculate the total processing time, one must model the dependence of each block on the number of antennas, subcarriers, and other block-specific parameters. In general, this can be daunting as the uplink chain contains numerous blocks: FFT/IFFT, channel estimator, equalizer, demapper, descrambler, rate dematcher, and Turbo decoder. However, we can construct a simple yet accurate model by making the following observations: (i) processing time of blocks that operate on the OFDM symbol level (e.g., FFT, equalization), including the memory copy, varies linearly[1] with the number, N, of antennas; (ii) the processing time of blocks using the constellation symbols (e.g., demapper, dematcher) is a function of the modulation order; (iii) the processing time of decoder is determined by the number of iterations (denoted by L), the subcarrier load D, and the observation that the decoder processes D bits per subcarrier in each iteration.

Thus, the total processing time can be written as:

$$T_{rxproc} = w_0 + w_1 \cdot N + w_2 \cdot K + w_3 \cdot D \cdot L + E, \quad (1)$$

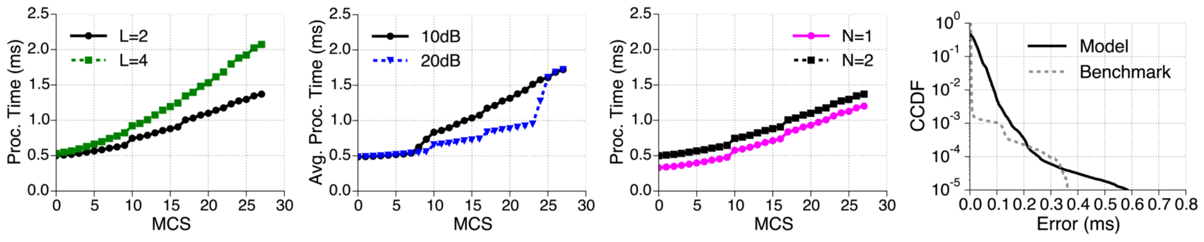
where $w_0,\ w_1$ and w_2 are constants; E is the error term that includes modeling error and the variability of the execution environment, and K is the modulation order of the MCS. The constants in Eq. (1) are largely implementation- and platform-specific, as they depend on type of optimizations (e.g., vectorization) as well as the architecture.

The processing time depends indirectly on the wireless channel condition through the number of iterations, L, that are required to decode a packet, i.e., pass the CRC checksum. To avoid excessive delay, receivers typically limit decoding to at most L_m iterations. Therefore, irrespective of the channel condition, we obtain an WCET bound on processing time by substituting L with L_m in Eq. (1). Note that the number, L, of iterations is in general non-deterministic (even for fixed SNR) and may take any value in $[1,\ L_m]$.

To validate our linear model, we collect data (see §4.2) on the total uplink processing time of a 10MHz LTE system (50 PRBs) for different MCS (0–27), SNR values (0–30dB), and different number of antennas. The maximum number, L_m, of Turbo iterations is set to 4. For each measurement, we note the load, D, and the iteration count, L, and then apply a linear regression to determine the model parameters.

Table 1 shows the model parameter estimates obtained from 4×10^6 measurements on a GPP platform. We also show the goodness-of-fit metric, r^2, in each scenario. The fitness metric is observed to be close to 0.99, indicating the

[1]Assuming maximum-ratio combining, the equalization complexity with spatial multiplexing is N^2 [32].



(a) vs. Iterations ($N = 2$) (b) vs. SNR ($N = 2$) (c) vs. Antennas (d) Error (E) distribution

Fig. 3: Plots showing the variations in processing time.

	w_0	w_1	w_2	w_3	r^2
GPP	31.4	169.1	49.7	93.0	0.992

Table 1: Model parameter estimates (in μs).

high accuracy of our model. Based on the model parameter estimates, we can observe, for instance, that each additional antenna adds 169μs while each Turbo iteration at MCS 27 adds 345μs.

Fig. 3 shows the total processing time as we vary the number of iterations, SNR, and the number of antennas. From the plots, it is evident that the processing time exhibits high variability. For instance, it increases by a factor of 2.8 (from 0.5ms to 1.4ms) as MCS changes from 0 to 27 (Fig. 3(a)). Further, the total time with four Turbo iterations compared to two iterations increases the total processing time by more than 0.5ms, which is consistent with observations made in previous studies [25, 35]. Note that in Fig. 3(b), decreasing the SNR from 20dB to 10dB increases the processing time by more than 50% between MCS 13 and 25. Similarly, for a fixed post-processing SNR, increasing the number of antennas to 2 adds nearly 200μs to processing time.

In summary, we find that wireless processing is a dynamic workload that varies with data rates, channel conditions and the number of antennas used.

Platform Error. Fig. 3(d) shows the distribution of the error between the model and the actual processing time. In 99.9% of observations, the error is less than 0.15ms. However, for a few measurements, the error can be as high as 0.7ms. Since the processing runs on a soft real-time system, the processing could be disrupted due to kernel tasks such as interrupt handling. The error term could thus be significant for some observations. Nevertheless, this could also be attributed to a large model error. To confirm otherwise, we perform a separate stress test on the processing platform. We ran the cyclictest [5] latency measurement tool alongside a benchmark load generated using hackbench [6]. The cyclictest was run with the highest system priority and was expected to show a near-constant latency. Fig. 3(d) shows the latency distribution from the benchmark. The mean latency is 0.2ms, but some of the measurements have a latency above 0.4ms. We also observe that the *order statistics* of the modeling error is roughly similar to the benchmark latency. For example, 1 in 10^5 measurements had a latency of more than few hundred microseconds. This confirms that the distribution of error term in Eq. (1) is mostly influenced by the platform and not by the model error.

2.2 Parallelism

While our model provides the total processing time, it does not show the processing times of individual blocks. For simplicity, we assume the processing chain comprises of three sequential tasks: FFT, *demod*, and *decode*, where the demod task comprises channel estimation, channel equalizer and constellation-demapper; and decode task includes rate-dematcher, de-scrambler, and Turbo decoder. Similar to Eq. (1), one can obtain a model for the processing time of each block.

Thus far, we have modeled processing time for a single thread running on a single core. However, it is possible to exploit different levels of parallelism, e.g., antenna-level, symbol-level and subcarrier-level parallelism, by making use of multiple CPU cores [35]. The FFT task that runs on each of the 14 OFDM symbols of each antenna is easy to parallelize. Similarly, channel equalization that runs on each OFDM symbol can be parallelized. Turbo decoding which is the most time-consuming operation can be parallelized over code-blocks, where decoding and CRC check can be done independently on each code-block [7]. For instance, at MCS 27, LTE utilizes 6 code-blocks all of which can be decoded concurrently.

Fig. 4 shows the processing times of FFT and decode tasks when it is parallelized over two cores. We can run FFT on 7 OFDM symbols on each core, and nearly halve the processing time (note the maximum overhead of 6μs). In the decode block, as seen in Fig. 4(b), parallelizing the Turbo decoding reduces the processing time by almost 310μs, from 980μs to 670μs.

Based on these observations, Fig. 5 shows a general breakdown of the processing of a subframe into tasks, and further into subtasks. For clarity, each task is shown to be completely parallelizable. For example, the FFT task can be parallelized by executing the FFT operation of each antenna's samples (an FFT subtask) on a different core. Another example is the equalization task, which can be performed independently (thus parallelized) for each group of subcarriers (an equalization subtask). Although a certain task might be parallelized, all of its subtasks must complete execution before moving on to the next stage. This establishes a de-

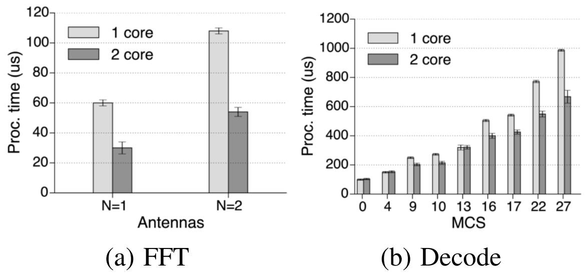


Fig. 4: Task execution times on multiple cores.

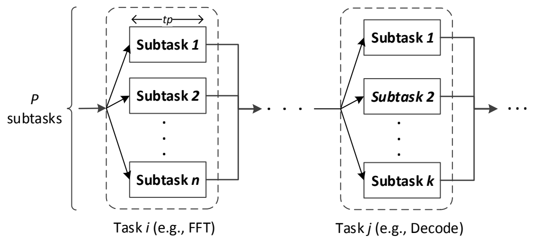


Fig. 5: Breakdown of subframe processing into tasks and subtasks.

pendency/precedence constraint between the different tasks involved in the subframe decoding. So, tasks have to execute in order to ensure correct decoding results. For the rest of the paper, we assume that the execution time of tasks (subtasks), except for the decoder, is deterministic.

2.3 Transport Latency

The transport of IQ samples from the radio front-ends to the cloud involves two separate networks as shown in Fig. 2. The fronthaul network, deployed using an optical fiber network, connects the radios to an optical switch located in the cloud. Various standards such as CPRI [4] have been proposed for transport over fronthaul networks. Also, a cloud network connects the optical switch to the pool of GPPs and VMs. The architecture of the cloud network is similar to a datacenter network (e.g., fat-tree topology) and includes aggregation and top-of-rack switches [19].

A wireless subframe incurs both fronthaul and cloud latencies. The fronthaul latency is a function of the length of the fiber (propagation time of light in fiber is approximately 5μs/Km) and the overhead of optical switching. While the exact fronthaul specifications are still under consideration, it is expected that the distance between remote radios and the cloud can be up to 20–40Km [2], resulting in a one-way propagation delay of 0.1–0.2ms excluding the overheads of (de)-packetization and cloud transport delay. While the fronthaul network has a fixed delay and almost negligible jitter [19], the cloud transport latency is less deterministic as it involves a mix of hardware, software and virtualized interfaces. To see the impact of the cloud network, we measure the one-way latency (measured from the round-trip time) between an external host and cloud resource. The host and the cloud resource are connected over 1/10 GbE Ethernet

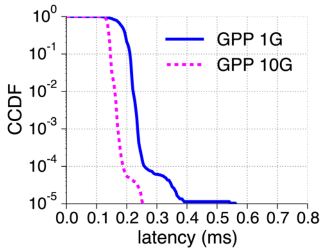


Fig. 6: Distribution of cloud network delay.

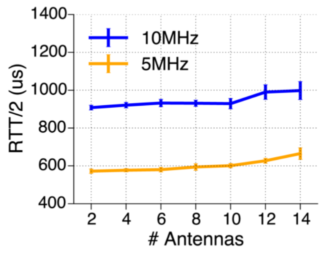


Fig. 7: One-way transport latency for GPP vs. number of antennas for 10GbE.

through a switch. We obtain the measurements by sending 1000 packets per second (LTE processes 1000 subframes per second) between the host and the cloud resource.

Fig. 6 shows the distribution of one-way cloud latency for GPP platform with 1 and 10Gbps Ethernet network, respectively. The mean transport latency is around 0.15ms. However, this latency has a long tail distribution where around one in 10^4 packets, for both 1GbE and 10GbE connections, has a latency more than 0.25ms. These observations imply that using the mean statistic of the transport latency is not good enough to provide latency guarantees.

To emulate C-RAN's transport network, we build a medium-size testbed of 16 WARPv3 radios that are connected using Ethernet to an off-the-shelf GPP. The radios are connected via 1 GbE port and then aggregated using a 1/10 GbE switch into GPP's 10GbE port. We use the CWARP transport library [20] to implement the read and write operations. Fig. 7 shows the one-way transport latency as we vary the number of antennas/radios and the bandwidth. In the 5 MHz case, we observe that the maximum latency is 620μs while it exceeds 1000μs (or 1ms) for 10MHz bandwidth. Since LTE frames arrive every 1ms, to prevent queuing delay, at most 8 antennas at 10 MHz can be supported on the GPP.

2.4 Deadline-miss

Once an uplink subframe is received at the radio frontends, an ACK or NACK response must be included in the downlink subframe that is transmitted exactly 3ms later. However, as illustrated in Fig. 8, not all of 3$ms$is available for Rx processing. For example, subframe N received by the basestation (after acquisition) in the uplink needs to acknowledged by subframe $N + 4$ in the downlink. The Tx processing that encodes the response subframe cannot wait indefinitely for the Rx to finish. We assume the Tx processing starts 1ms be-

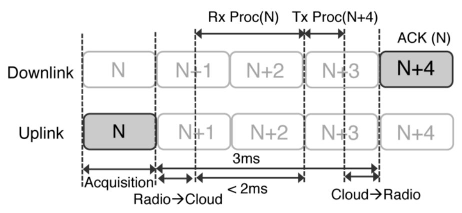


Fig. 8: Example sequence of subframes in LTE showing the Rx and Tx processing timelines.

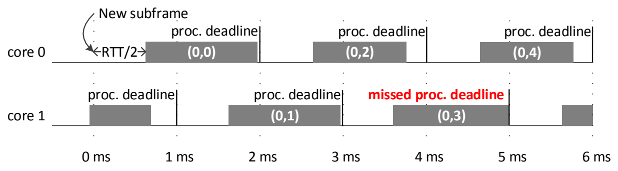


Fig. 9: An example of a partitioned schedule on two cores. Notation (i,j) refer to processing the j^{th} subframe of the i^{th} basestation.

fore the actual over-the-air transmission of downlink frame[2]. As a result, only $2ms$ is effectively available for Rx processing, which includes the transport delay from radios to the cloud.

Formally, we can express the end-to-end timing requirements as:

$$T_{rxproc} + \overbrace{T_{fronthaul} + T_{cloud}}^{\text{RTT/2}} \leq 2ms \qquad (2)$$

where $T_{fronthaul}$ is the fixed fronthaul latency and T_{cloud} is the cloud network latency. For ease of notation, the combined transport latency is denoted by RTT/2. One can rewrite Eq. (2) using the model given in Eq. (1). Furthermore, we can use the resulting equation to calculate the probability of a deadline-miss event. Assuming the fronthaul latency is fixed, the calculation requires the underlying distribution of the network latency and platform error. Since the exact distribution is difficult to model, we can use the empirical distribution obtained from separate network and stress tests.

3. C-RAN SCHEDULING

In what follows, we discuss the different approaches to scheduling subframes in C-RAN. We also present the design and implementation of RT-OPEX, a novel scheduling approach that reduces the deadline-miss rate in wireless frame processing.

A typical C-RAN compute resource consists of two main components: transport and processing. The transport component makes an LTE subframe available every $1ms$ for processing. The processing component, on the other hand, receives these subframes and attempts to decode each of them within the available processing time budget. Specifically, the processing time of each subframe, T_{rxproc}, should be less than the processing budget T_{max}, such that

$$T_{rxproc} \leq T_{max} := 2ms - (RTT/2). \qquad (3)$$

The compute resource in a C-RAN can be viewed as a multiprocessor host executing a set of processing threads, with each thread continuously running on a single core. The role of a scheduler is then to assign tasks for each processing thread, where a task represents a subframe processing event. Its design takes as inputs the number of available processing cores along with the number of assigned basestations.

[2]Consistent with OpenAirInterface [8] implementation

3.1 Conventional Scheduling Approaches

There are two types of schedulers: *partitioned* and *global*. Described below are their design and implementation.

3.1.1 Partitioned Scheduler

Partitioned schedules are usually determined offline; each incoming subframe is assigned to a core based on a predetermined schedule. Such schedules have the advantages of providing statistical real-time processing guarantees and generating deterministic schedules.

To design the partitioned schedule, we utilize the processing budget T_{max} (note that $T_{max} < 2ms$) as the execution time of each subtask. If the actual execution time exceeds T_{max}, then a deadline-miss will occur. The scheduler assigns $\lceil T_{max} \rceil$ cores for each basestation. Assuming there are M basestations to schedule the processing of subframes, for a basestation of id i ($0 \leq i \leq M-1$), it maps the subframe of index j to core $i * \lceil T_{max} \rceil + j \ mod \ \lceil T_{max} \rceil$.

As a basestation receives a new subframe each $1ms$, the partitioned scheduler schedules two subframes on the same core each $\lceil T_{max} \rceil \ ms$. This guarantees each subframe to have $\lceil T_{max} \rceil \ ms$ of available processing time on the core it is assigned to, which is larger than its upper bound of T_{max} ms. Fig. 9 shows an example partitioned schedule on a 2-core host (the deadline-miss event in the figure will be discussed in Section 3.2). In this example, $\lceil T_{max} \rceil = 2$, so the partitioned scheduler assigns the subframes in a round-robin fashion on the two cores each $1ms$.

3.1.2 Global Scheduler

We utilize a single queue shared across basestations to implement the global scheduling of subframes on a single computing node. The queue is realized with a fixed-size ring-buffer that holds the incoming subframes from the basestations. A scheduling thread runs on a separate core and dispatches subframes from the queue to the available cores (each running a processing thread) for processing according to EDF schedule. Note that EDF is equivalent to FIFO scheduling when all basestations have the same transport delay, since all subframes have the same deadline T_{max}.

Each core will process at most one subframe at a time. If the processing does not end before the deadline, the processing thread terminates the ongoing task and goes to an idle state. It then waits for the next dispatched subframe. Fig. 10 shows an example global schedule of 2 basestations on a 2-core host. In this example, when the processing finishes

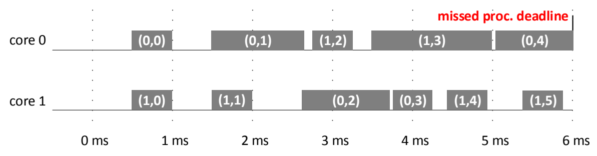


Fig. 10: An example of a global schedule of two basestations on two cores.

before the next subframe arrives, no queuing takes place, as seen at $t = 1ms$. On the other hand, at $t = 4ms$, the 4^{th} subframe of basestation 0, is queued and only dispatched at $t = 5ms$, so that it misses its processing deadline (at $6ms$).

Next, we present RT-OPEX, which builds on top of a partitioned scheduler and utilizes idle processor cycles to reduce deadline-misses.

3.2 RT-OPEX

Three characteristics define the subframe processing tasks — they are *periodic*, have *non-deterministic execution times*, and require *real-time constraints* to be met. Due to the non-deterministic nature of wireless frame processing, a partitioned scheduler cannot account for deadline-misses or idle times of the cores at design time. Moreover, subtasks comprising a task (e.g., Turbo decoding) exhibit highly dynamic execution times. A partitioned scheduler considering each subtask as an independent execution unit still suffers from the same issues of potentially missing deadline and idle cores (in addition to the complex scheduler design from precedence constraints).

For example, as shown in Fig. 9, the processing time, T_{rxproc}, of a frame might take time less than $\lceil T_{max} \rceil$, so that the core will be idle for the amount of time equal to $\lceil T_{max} \rceil - T_{rxproc}$. On the other hand, in extreme cases, T_{rxproc} will be larger than T_{max}. This event will force a scheduler to drop the subframe to ensure the schedulability of incoming subframes, causing a deadline-miss. Thus, the processing thread (running at each core) alternates between two states: *active* and *waiting*. The active state corresponds to the case when it is processing a subframe (darkened portions of Fig. 9). The waiting state, on the other hand, corresponds to the case when it is not performing any active processing (empty portions of Fig. 9).

3.2.1 RT-OPEX Design

The design of RT-OPEX is inspired by an intuitive observation: if the processing thread of core 1 in Fig. 9 were able to utilize the idle cycles of core 0, then it would not have missed its deadline.

RT-OPEX **op**portunistically **ex**ecutes a portion of a processing task on another idle core, which we refer to in the rest of this paper as "migration". RT-OPEX is independent of any partitioned scheduler employed underneath. As long as multiple processing threads are running on different cores, there will be time intervals during which the active and waiting states of these threads will overlap. RT-OPEX exploits

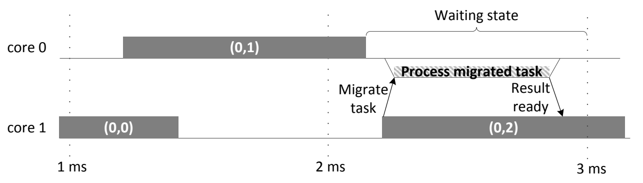


Fig. 11: An example scenario of RT-OPEX showing migration between two cores.

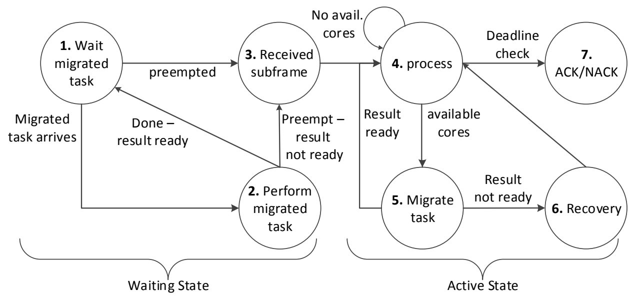


Fig. 12: The state diagram of the processing thread in RT-OPEX.

this phenomenon to decrease processing time, reduce the deadline-miss probability, and improve performance.

A. High-Level Description: At a high level, RT-OPEX *migrates* a subtask from a processing thread in its active state to another processing thread (running on a different core) in the waiting state. In the rest of this paper, we refer to the subframe processing task assigned to a processing thread by the scheduler as the *processing task*, and the subtask (part of the processing task) migrated by RT-OPEX to another core as the *migrated subtask*.

Fig. 11 shows an example of migration. At $2.3ms$, RT-OPEX finds that the processing thread on core 0 is in its waiting state. So, it migrates a subtask from the processing task of core 1 to run on core 0. At that point of time, the processing task will be executing in parallel on both cores (with independent subtasks executing on both cores). After the migrated subtask finishes execution, it makes its result ready for the processing thread to consume it. The processing task then completes execution on core 1 and the processing thread on core 0 returns to its waiting state.

B. Migration Mechanism: Fig. 12 shows the state diagram of RT-OPEX. A processing thread alternates between two states, active and waiting. In its active state, it executes the processing task, and might execute migrated subtask(s) in its waiting state.

1. Waiting State: When the processing thread is in its waiting state, it waits for a migrated subtask for execution (from another processing thread — state 1). When such a subtask arrives, the processing thread starts executing the migrated subtask immediately (state 2), where two of the following events might happen.

1. The migrated task completes before a new processing task is available, i.e., before it gets preempted. RT-

Algorithm 1 The Migration Algorithm in RT-OPEX.

```
 1: Input: $P$ subtasks, each subtask has $t_p$ proc. time.
 2: Input: $R$ cores, each core has $fc_k > 0$ of free time.
 3: Input: $\delta$, cost of migrating a subtask to another core.
 4: $S \leftarrow P$                ▷ S is # of left subtasks (not migrated)
 5: $max_{off} \leftarrow 0$        ▷ max # of migrated subtasks per core
 6: while $S > 1$ and $k \leq R$ do
 7:     $lim_{off} = \lfloor \frac{fc_k}{t_p+\delta} \rfloor$    ▷ # of subtasks can be migrated
 8:     $n_{off} \leftarrow min(S - max_{off}, lim_{off}, \lfloor \frac{S}{2} \rfloor)$
 9:     $max_{off} \leftarrow max(n_{off}, max_{off})$
10:     Migrate $n_{off}$ subtasks to $k^{th}$ core
11:     $S \leftarrow S - n_{off}$
12:     $k \leftarrow k + 1$
13: end while
```

OPEX sets a *result ready* flag for the "remote" processing thread (the one migrating a subtask) to consume the result. The thread then returns to waiting for a migrated subtask (state 2 $\rightarrow$ state 1).

2. The migrated subtask is preempted at the deadline before it is completed. This indicates that a new processing task is available for the processing thread. RT-OPEX sets a *result not ready* flag and switches the processing thread to the active state (state 2 $\rightarrow$ state 3).

While waiting for a migrated subtask (at state 1), the transport component can preempt the processing thread to indicate that a new processing task is available (state 1 $\rightarrow$ state 3).

2. Active State: When the processing thread receives a new processing task, it switches to the active state, and starts processing the subframe (state 4).

The processing thread starts processing the subframe until it reaches a parallelizable task which offers an opportunity for migration to idle cores (such as FFT or decoder). As the arrival of subframes is deterministic, the underlying scheduler should be able to inform when each idle core will be preempted and switched to active processing. As such, the scheduler can compute the potentially available time budget for migration on each idle core.

RT-OPEX uses this knowledge along with the model of the subtask execution time to decide how many subtasks to migrate to each core (state 4 $\rightarrow$ state 5) by applying Alg. 1. RT-OPEX follows a greedy approach; it tries to migrate subtasks as much as possible with one caveat. The time to execute local subtasks (i.e., those that are not migrated) should be larger than the maximum time of executing migrated subtasks at each idle core (including the migration overhead). This serves to satisfy one important requirement of RT-OPEX. The performance of RT-OPEX must be equal to or strictly better than the case without use of migration. By the time the processing thread finishes executing the local subtasks, all the migrated subtasks must have finished execution so that it can combine their results without any delay. It is worth noting that, in our model, we associate each subtask (comprising a parallelizable task) with a fixed and deterministic execution time. So, we can treat each subtask as a single execution unit. In particular, the number of migrated subtasks to a core k, n_{off}, should satisfy the following requirements.

R1. It must be less than the maximum number, k. of subtasks core can accommodate (given in line 7 as lim_{off}), such that $n_{off} \leq lim_{off}$. lim_{off} includes the subtask execution time as well as the subtask migration cost: δ, such that: $n_{off} \leq \lfloor \frac{fc_k}{t_p+\delta} \rfloor$, where fc_k is the available time budget at core k, and t_p is the subtask execution time (computed from our model). δ includes costs incurred during subtask migration such as cache thrashing and accessing the CPU states.

R2. The number of subtasks left after migrating $S - n_{off}$ should be larger than the maximum number of subtasks already allocated to any other core such that $S - n_{off} \geq max_{off}$. It follows that $n_{off} \leq S - max_{off}$.

R3. The number of un-migrated subtasks, $S - n_{off}$, should be larger than the number of subtasks migrated to core k. We need this condition as the previous step doesn't count in the subtasks to be migrated to core k. We then have $S - n_{off} \geq n_{off}$ so that $n_{off} \leq S/2$.

In line 8, Alg. 1 combines the three requirements so that $n_{off} = min(S - max_{off}, lim_{off}, \lfloor \frac{S}{2} \rfloor)$. After calculating n_{off}, RT-OPEX migrates n_{off} subtasks to core k. It repeats the same process until either the number of subtasks for migration or the number of cores is exhausted.

Thanks to Alg. 1, the processing thread does not wait for any migrated subtask for completion. By the time the processing thread finishes the local subtasks, all migrated subtasks are completed in the ideal case. Each migrated subtask is associated with a flag that indicates its completion status (*result ready* vs. *result not ready*). The processing thread checks this value to decide whether to engage the recovery procedure or not. If all migrated subtasks have completed execution, the processing thread can use their results and move on with the execution (state 5 $\rightarrow$ state 4). On the other hand, if the local processing is complete and at least one migrated subtask is not completed (has its flag set to *result not ready*), the processing thread goes to recovery state (state 5 $\rightarrow$ state 6).

Execution of the migrated subtask can be incomplete because its execution can take longer than anticipated due to background and other kernel processes. The recovery state handles the case of such inaccurate migration decisions by RT-OPEX. It involves computing the results for those incomplete migrated subtasks. This ensures that the performance of RT-OPEX will be no worse than the baseline case. In the baseline case, all subtasks are executed serially which corresponds to the worst-case scenario of RT-OPEX (no migrated subtask completed execution).

After all subtasks corresponding to a single processing task are completed, the processing thread continues executing the rest of the decoding tasks. It repeats the same procedure for any task that can be parallelized. RT-OPEX always monitors whether the processing thread violated the task's processing deadline ($T_{rxproc} > T_{max}$). Depending on the deadline check status, the processing might result in either an "ACK" or "NACK" message to the radio (state 7) after execution has completed. RT-OPEX then switches the processing thread back to the waiting state.

4. IMPLEMENTATION AND EVALUATION

We implement RT-OPEX and rest of the schedulers on a testbed of software radios, Ethernet, and commodity server hardware. In this section, we give the details of the implementation, the evaluation platform, and performance evaluation of the schedulers.

4.1 Implementation

We have built a multiprocessor scheduler for a single computing resource of a C-RAN from the ground up. Our scheduler utilizes the low-level `pthread` library to implement the processing and transport components. Each thread is mapped to a single kernel-level thread (1:1 mapping), and is bound to a single processing core (overrides OS thread-scheduling). The transport threads run on a dedicated set of cores that are separate from processing cores. The transport and the processing threads are synchronized using semaphores. As the transport threads are critical to maintaining synchronization between the radios and the GPP (triggered every $1ms$), we use a *one-way locking* mechanism where processing threads wait for the transport threads (not the other way around). The processing threads are signaled on two occasions: (1) when the transport threads finish writing to the sample IQ buffer indicating the arrival of a new subframe, and/or (2) when a migrated subtask is available from another processing thread. We implement a common watchdog timer that maintains a global reference time that allows detecting deadline-misses across the cores. We also implement a shared data structure, indexed by each core ID, to maintain the CPU states (active, idle — with remaining time) that each processing thread (of each core) updates and polls.

Our scheduler integrates with the OpenAirInterface (OAI) PHY library [8]. OAI is an open-source software implementation of LTE that includes both RAN and Evolved Packet Core, and implements all the PHY-layer functions of LTE Rel 10. OAI implements its out-of-the-box partitioned scheduler for uplink and downlink processing, but it is not amenable to PHY-layer migration. Therefore, we modularize the OAI processing and write an abstraction layer that abstracts the PHY functions at task level, labeled as `taskX`, and subtask level, labeled as `subtaskX`, where `X` $\in$ {FFT, demod, decode}. Each of the subtasks can be executed independently, and thus, provides the basis for parallelism. Since OAI implements a complete baseband chain, our abstraction code is tested using traces from OAI simulators to make sure that the processing is reproducible. This step was essential to ensure the correct functioning of the scheduler when the OAI data structures are duplicated for multiple basestations.

Using our system design, partitioned scheduling is realized by fixing the threads on which basestation's subframes are processed, i.e., when a particular subframe is received from the transport, only the corresponding processing thread is notified. In case of global scheduling, there is no binding of a basestation to threads, and any idle processing thread can process an arriving subframe. In RT-OPEX, the migration of subtasks implies that the some of the `subtaskX` routines from the current processing thread are migrated to a different processing thread. As the global OAI variables and the baseband samples are held in a shared memory (L3 or main), migration of data is realized by passing the references to the memory contents.

Enforcing deadlines. For correct operation of a real-time system, we must ensure that a frame processing task must be completed before, or terminated at its deadline. Implementing this in our system is challenging since we abstract away the low-level PHY routines. One possible solution is to pass a timer to each OAI function, and constantly check on the timer. This approach, however, is not practical. Instead, we check on the slack time (defined as the available time before the task's deadline) before we execute each task; using our task model, we check if the execution time is less than the slack time, else we drop the task and the subframe. The resulting gaps are, however, not used for migration.

Our implementation code is packaged as an open-source tool that evaluates the performance of different schedulers. Fig. 13 depicts the implementation framework used in our evaluation. The tool enables evaluations on different operating systems, including thread scheduling models (e.g., Round-Robin and FIFO), real-time kernel, virtualization, etc. Ultimately, the tool can be used to profile the system performance (deadline-miss rate, load, memory usage) which can, in turn, help operators design and provision compute resources for C-RAN.

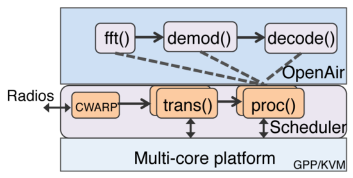


Fig. 13: Implementation framework for RT-OPEX.

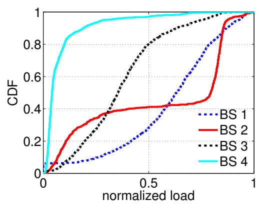


Fig. 14: Basestation load distribution.

4.2 Evaluation Platform

We use different state-of-the-art computing and networking platforms in our evaluations. For computing, we use a general commodity server (i.e.,GPP), a 32-core (hyper-threading enabled) machine with Intel Xeon E5-2660 2.2 GHz x86 CPUs (SandyBridge architecture), 128GB RAM, 15MB L3 cache, and 1/10 GbE Ethernet ports. The evaluation with virtualization platforms such as containers is left to future work. To closely match the performance of datacenter networks, we consider 1 GbE and 10 GbE Ethernet links that are connected to the GPP through an HP 6600 series Ethernet switch using standard Intel network drivers.

Optimizations. Several optimizations are applied to get the best computing performance. The OAI workload runs on an Ubuntu 14.04 low-latency kernel. Considered as a soft real-time system, the low-latency version is a stable kernel compared to other hard real-time kernels like RTLinux [10] which require custom patches. Various optimization features, such as SSE3/SSE4 instruction set, O3 flags, etc., are enabled. Further, the power-saving features and sleep states available on Intel processors are disabled. This ensures the CPU cores run at a constant maximum frequency. To minimize disruptions from interrupts, the processing threads are pinned to dedicated cores and use FIFO scheduling. We use the OAI timestamps to calculate the processing time.

Data collection. Since publicly available basestation traces are difficult to obtain, we devise a measurement setup to get the variations of cellular traffic. We use USRP software radios and log RF samples off the air on Band-13 and Band-17 LTE downlink channels in a city environment. Specifically, we log the signal of 4 cellular towers and estimate the load by correlating with the average signal energy every $1ms$. Fig. 14 shows the distribution of the load variations for the 4 basestations. We then ran the scheduler with IQ traces from OAI, where the MCS of each subframe is determined by our basestation load trace. For each experimental setting, we use an AWGN channel model with a fixed SNR of 30dB (and varying MCS according to load) and collect the processing logs of 30000 LTE subframes from each basestation.

Experimental setup. We consider a 4-basestation setup, each with two antennas ($N = 2$), running on a GPP platform. We specify the OAI LTE bandwidth to 10MHz, which corresponds to a sampling rate of 15.36MHz, i.e., each subframe contains 15360 samples. We consider a single user uplink transmission and assume 100% PRB utilization. It is worth noting that 100% PRB utilization constitutes a conservative scenario of a single user for all subframes. This reduces, on average, the opportunities of migrations (resulting in lower performance gains) as compared to a realistic scenario with multiple users and varying PRB utilization. As we were not able to locate decodable real-world BS traces for multiple users, we opted to emulate the BS uplink traffic load through MCS variations and assumed a single user generates that load. Under this configuration, the nominal PHY throughput can vary from 1.3 to 31.7Mbps, depending on the MCS used. Further, we choose $\lceil T_{max} \rceil = 2$, i.e., each basestation is assigned 2 CPU cores under partitioned scheduling.

We first run the processing with our C-RAN testbed with radios and the Ethernet transport. As mentioned earlier, the radios in our testbed are WARPv3 SDR boards. From Fig. 7(a), observe that the one-way latency from radios to the GPP at 10MHz bandwidth is as high as $0.9ms$. This effectively leaves $1.1ms$ to process each subframe (which is much less than the processing time of nearly $1.5ms$ at MCS 27), resulting in a very high deadline-miss rate. Therefore, to accurately emulate real C-RAN deployments, we replace the WARP transport with a fixed transport delay (RTT/2) value ranging from $0.4ms$ to $0.7ms$, that represents various off- and on-site deployment scenarios. In what follows, we describe the C-RAN performance of a GPP platform made up of a single physical machine.

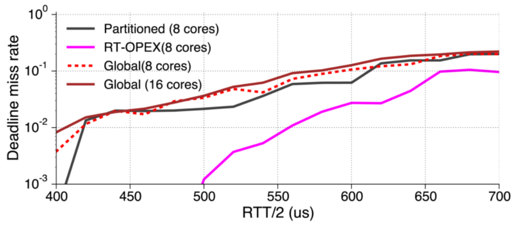


Fig. 15: Deadline-miss comparison of schedulers.

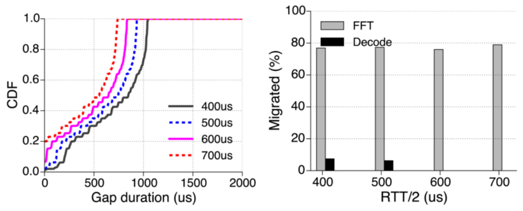


Fig. 16: Gaps and migrations in RT-OPEX.

4.3 Performance comparison

For each transport delay setting, we evaluate the deadline-miss rate for the four basestations and for each scheduler. Fig. 15 shows the deadline-miss performance of the different schedulers. The main takeaways from the figure are as follows.

RT-OPEX **Performance:** RT-OPEX exhibits virtually zero deadline-miss rate when latency is less than $500\mu s$. To understand this further, let's look at Fig. 16. For RTT/2 less than $500\mu s$, the partitioned scheduler has gaps (only due to processing time variation) larger than $500\mu s$ for 60% of the processed subframes. RT-OPEX utilizes these gaps to migrate FFT and decoding subtasks as evident from the right plot of Fig. 16, where 20% of the decode subtasks are migrated. These migrated decode subtasks belong to subframes with high MCS that are responsible for the deadline misses in the original partitioned scheduler. By migrating these tasks, the processing time drops well below $1500\mu s$, which is less than the processing budget.

As latency increases beyond $500\mu s$, the gaps get narrower, thus reducing the chances for migrating the decode subtasks. Nevertheless, RT-OPEX keeps on migrating the smaller size FFT subtasks, resulting in the deadline-miss rate significantly lower than that of the partitioned and global schedulers. As evident from Fig. 15, the deadline-miss rate of RT-OPEX is one order-of-magnitude better ($10^{-2} \rightarrow 10^{-3}$) than that of both partitioned and global schedulers.

Partitioned Scheduler: Unlike RT-OPEX, a partitioned scheduler can't exploit gaps available because of the processing time variations. This is evident from the sudden rise of the deadline-miss rate when RTT/2 exceeds $400\mu s$. The available time budget of processing falls below $1600\mu s$. Referring to Fig. 3(a), the processing time can exceed $1.5ms$ for higher MCS values.

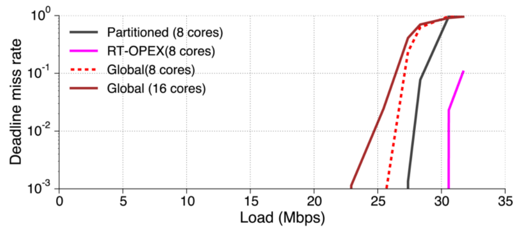


Fig. 17: Deadline-misses vs. load (RTT/2=500μs).

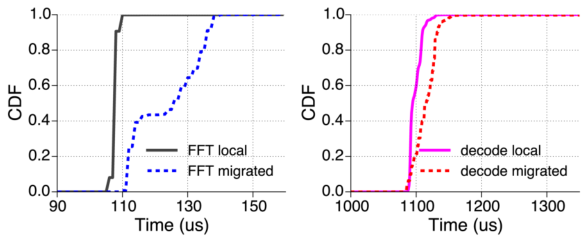


Fig. 18: Comparison of processing times of local and migrated tasks.

As a result, most subframes with MCS larger than 20 will miss their processing deadlines. As RTT further increases above 400μs, the deadline-miss rate increases, albeit at a slower rate. Subframes with lower MCS values can be successfully decoded within $1.3ms$ (corresponding to RTT/2 = 700μs).

Global Scheduler: The global scheduler exhibits the most surprising behavior. We evaluate two global schedulers, one running with 8 cores while the other utilizes 16 cores. Theoretically, this scheduler should perform as good as a partitioned scheduler. Both schedulers provide a subframe with all the time needed to finish decoding before its deadline.

Nevertheless, as evident from Fig. 15, the global scheduler (1) performs slightly worse than the partitioned scheduler, (2) does not improve when the number of cores doubles from 8 to 16, and (3) does not exhibit a zero deadline-miss rate even at the lowest RTT value. As explained later, several factors related to the design and execution of the global scheduler contribute to this surprising phenomenon.

In Fig. 17, we set RTT/2 to 500μs and show the deadline-miss performance for different subframe loads (corresponding to different MCS values). RT-OPEX's gains are shown to be prominent at higher loads (30Mbps and above) where rest of the schedulers miss deadlines for 100% of the frames. Therefore, assuming a deadline-miss threshold of 10^{-2} that is typical of real-time systems, RT-OPEX can support 15% higher load (31Mbps compared to 27Mbps) than a default partitioned scheduler.

4.4 Migration Overhead

The major overhead in migration comes from the transfer of contents from shared memory. The additional overhead of maintaining CPU states via a shared data structure is negligible. Given the complex memory hierarchy of modern processors, providing a detailed cache analysis is outside of the scope of this paper. However, we use our abstraction to calculate the overhead of task migration by simply measuring the time it takes to process a local and a migrated task. Fig. 18 compares the processing times of the tasks that are performed locally and that are migrated to and executed on a different core at runtime. Observe that there is always a non-zero overhead to migrate a task. For example, for FFT, the median processing time increases from 108μs to 126μs when it is migrated, i.e., an 18μs increase. For the decode task, the overhead is nearly the same at 20μs. Thus, the cost of migration is a fixed across the subtasks, which corresponds to the fetching of global OAI variables from shared memory to on-chip/local memory. Note that the subframe buffer and transport block buffer are both referred within the OAI eNB data-structure, and therefore, both FFT and decode migration (including subtask) involve the same amount of memory transfer.

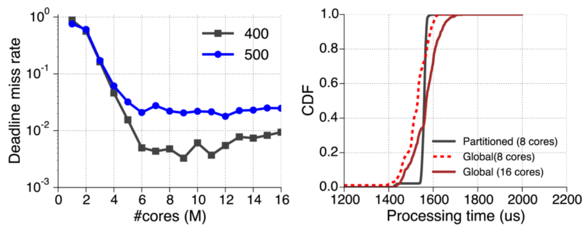


Fig. 19: Global scheduler as cores are varied.

Global scheduling, while offering increased flexibility, suffers from high overheads. More interestingly, increasing the number of cores does not mitigate this (and even decreases performance). To see this, consider Fig. 19, where deadline-miss performance saturates and even worsens beyond eight cores. This is attributed to increased cache thrashing. Each core in global scheduling processes different basestations every few subframes, which leads to frequent flushing of its memory cache and adds to the processing times, as evident from the right plot in Fig. 19, which shows the processing time distribution for MCS 27. From the plot, we observe that global with 16 cores has a considerably larger processing time (80μs) for more than 10% of the subframes.

5. DISCUSSION

Based on our evaluation of the different scheduling approaches, we now discuss them in the context of operator deployments.

A. Overhead: The additional scheduling overhead of a partitioned schedule is minimal; each subframe is assigned to a predetermined core without the need of locking or migration across cores. On the other hand, a global scheduler incurs higher overhead because of frequent cache trashing. RT-OPEX incurs the overhead of subtask migration, which we estimated to be in the order of 20μs for both FFT and decode subtasks. RT-OPEX takes this overhead into account while migrating so as to guarantee feasible migration. Even with the overhead, we show in §4.3 that RT-OPEX achieves a significant improvement in deadline-miss performance.

Table 2: Qualitative comparison of related scheduling approaches in C-RAN.

	Migration	Compute Resources	Granularity
PRAN [31]	Yes	Dynamic	Subtask
CloudIQ [15]	No	Fixed	Task
WiBench [34]	No	Fixed	Subtask
BigStation [32]	No	Fixed	Subtask
RT-OPEX	Yes	Fixed/Dynamic	Subtask

B. Flexibility to resources: Available resources in C-RAN might change over time as storage, memory, and processor failures are common in a datacenter running on commodity hardware [30]. As a partitioned schedule is provisioned to a set of fixed resources, any change in the available resources results in a significant performance degradation [16]. Alternatively, a global schedule, by virtue of its design, adapts to the underlying resources without the need to design a new schedule (Fig. 19). RT-OPEX suffers from the same limitation of a partitioned schedule, but can automatically exploit any added resources to migrate subtasks.

C. Flexibility to load: From our measurements, we observe that processing times exhibit millisecond-level variations due to varying traffic loads and channel conditions (Fig. 1). However, the partitioned schedule fails to adapt to varying processing times. When the processing time of a subframe exceeds the deadline, partitioned schedules drop the subframe resulting in a deadline miss. This occurs even though processing on another resource might introduce a gap. RT-OPEX fills the scheduling gaps by migrating subtasks to the available cores. It, therefore, adapts to the variations in the load. By design, the global scheduler is inherently flexible to the varying processing time.

D. Generality: The applicability of RT-OPEX is more significant for resource pooling in C-RAN where multiple basestations are processed together on a common platform. Particularly, for a heterogeneous set of basestations and standards (e.g., cellular-IoT [3]) where the traffic and channel conditions vary widely across the basestations, RT-OPEX can easily leverage idle cycles to improve performance.

6. RELATED WORK

Real-time wireless frame processing using software radios has been an active area of research over the past decade [25, 29]. Today, there are commercial software implementations of a fully functional LTE basestation [1]. However, their performance is nowhere close to dedicated hardware platforms. With the introduction of C-RAN [2], a slightly different variant — cloud-based processing — has received considerable attention. Some of the recent work in C-RAN has focused on its real-time implementation and scheduling. In Table 2 we summarize the properties of such approaches and show how RT-OPEX compares to them.

CloudIQ [15] provides a statistical framework to schedule multiple basestations on a multi-core platform in order to meet their processing requirements. However, it assumes fixed processing time (equal to the WCET) for each LTE subframe and does not take into account the idle compute cycles generated at smaller timescales. Moreover, CloudIQ treats each subframe processing task as an atomic execution unit, generates schedules for a fixed set of resources and provides no provisions for task migration. PRAN [31], a more flexible approach to resource management, proposes a pool of shared compute resources based on the dynamics of the load. PRAN further breaks each processing task into a set of subtasks, and allows for subtask migration across CPU cores. Nevertheless, PRAN's scheduling decisions are made before wireless frames are received, and thus cannot account for processing time variations due to channel conditions.

WiBench [34] is an open source framework that allows benchmarking of the wireless processing tasks. The authors analyzed the performance of LTE uplink processing to conclude that hardware acceleration is required for subtasks such as Turbo decoding. In a follow-up work [35], the authors present a system utilizing four GPUs to achieve real-time LTE subframe processing. The WiBench framework assumes fixed compute resources and does not support task migration. Finally, BigStation [32] provides an architecture for processing MU-MIMO frames in real time. It is based on parallelizing the processing subtasks of MU-MIMO and running them on a compute cluster made of commodity hardware. BigStation, similar to WiBench, assumes a fixed number of resources at runtime and provides no task migration.

RT-OPEX is different from the above solutions in that it supports migration at the subtask level. As the scheduling takes place at runtime, it works with both, fixed and dynamic nature of compute resources. It can be viewed as a specific application of work-stealing [17], which is a well-studied dynamic load-balancing technique for scheduling of parallel tasks.

The role of virtualization in RAN was described in [23, 25, 33]. A container approach to virtualization was shown to have a slightly better performance than a hypervisor approach. In [21], the authors adopt a two-tier model to manage a RAN where part of the control that requires less frequent changes goes to a central controller. The design of a remote radio head and its synchronization with the processing units was described in [36]. Finally, [19] and the references therein provide a comprehensive background on the state of the current C-RAN technology.

7. CONCLUSION

C-RAN is a promising solution to the problem of economically managing the scale of wireless processing in cellular networks. However, meeting frame processing deadlines without over-provisioning resources remains a major challenge. We proposed to meet this challenge with a new scheduling framework that builds on top of partitioned scheduling and opportunistically exploits the idle processing cycles for parallel processing of frames. Our evaluation results have demonstrated its potential in reducing deadline-misses at no additional cost.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. The work reported in this paper was supported in part by the NSF under Grants CNS-1160775 and CNS-1317411.

8. REFERENCES

[1] Amarisoft. http://amarisoft.com/. [Online; accessed 29-Jan-2016].

[2] C-RAN: The road towards green RAN. http://labs.chinamobile.com/cran/wp-content/uploads/2014/06/20140613-C-RAN-WP-3.0.pdf. [Online; accessed 29-Nov-2015].

[3] Cellular networks for massive IoT. http://www.ericsson.com/res/docs/whitepapers/wp_iot.pdf. [Online; accessed 10-Oct-2016].

[4] CPRI Specification V6.1. http://www.cpri.info/downloads/CPRI_v_6_1_2014-07-01.pdf. [Online; accessed 29-Nov-2015].

[5] Cyclictest. https://rt.wiki.kernel.org/index.php/Cyclictest. [Online; accessed 29-Nov-2015].

[6] Hackbench. http://manpages.ubuntu.com/manpages/trusty/man8/hackbench.8.html. [Online; accessed 29-Nov-2015].

[7] LTE Physical Layer Procedures. http://www.etsi.org/deliver/etsi_ts/136200_136299/136213/08.08.00_60/ts_136213v080800p.pdf. [Online; accessed 29-Nov-2015].

[8] Open Air Interface. http://www.openairinterface.org/. [Online; accessed 29-Nov-2015].

[9] RT-OPEX. https://github.com/gkchai/RT-OPEX. [Online; accessed 10-Oct-2016].

[10] RTLinux. http://rt.wiki.kernel.org/. [Online; accessed 29-Jan-2016].

[11] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessor. In *Proceedings of IEEE Real-Time Systems Symposium*, rtss, pages 193–202, 2001.

[12] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In S. H. Son, I. Lee, and J. Y.-T. Leung, editors, *Handbook of Real-Time and Embedded Systems*. Chapman Hall/ CRC Press, 2007.

[13] J. Baro, F. Boniol, M. Cordovilla, E. Noulard, and C. Pagetti. Off-line (optimal) multiprocessor scheduling of dependent periodic tasks. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1815–1820, 2012.

[14] A. Bastoni, B. Brandenburg, and J. Anderson. Is semi-partitioned scheduling practical? In *Real-Time Systems (ECRTS), Euromicro Conference on*, pages 125–135, 2011.

[15] S. Bhaumik, S. P. Chandrabose, M. K. Jataprolu, G. Kumar, A. Muralidhar, P. Polakos, V. Srinivasan, and T. Woo. Cloudiq: A framework for processing base stations in a data center. In *Mobicom*, 2012.

[16] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 152–160, Dec 1988.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of ACM*, 46(5):720–748, 1999.

[18] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press LLC, 2004.

[19] A. Checko, H. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. Berger, and L. Dittmann. Cloud ran for mobile networks – a technology overview. *Communications Surveys Tutorials, IEEE*, 17(1):405–426, 2015.

[20] K. C. Garikipati and K. G. Shin. Improving transport design for warp sdr deployments. In *Proceedings of the 2014 ACM Workshop on Software Radio Implementation Forum*, SRIF '14, 2014.

[21] A. Gudipati, D. Perry, L. E. Li, and S. Katti. Softran: Software defined radio access network. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, 2013.

[22] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982.

[23] C. Liang and F. Yu. Wireless network virtualization: A survey, some research issues and challenges. *Communications Surveys Tutorials, IEEE*, 17(1):358–380, Firstquarter 2015.

[24] R. G. Maunder. A fully-parallel turbo decoding algorithm. *IEEE Transactions on Communications*, 63(8):2762–2775, Aug 2015.

[25] N. Nikaein. Processing radio access network functions in the cloud: Critical issues and modeling. In *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, 2015.

[26] Y. Oh and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, 1995.

[27] Z. Shi, E. Jeannot, and J. J. Dongarra. Robust task scheduling in non-deterministic heterogeneous computing systems. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10, Sept 2006.

[28] A. Srinivansan and S. K. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84:93–98, 2002.

[29] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora:

High performance software radio using general purpose multi-core processors. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 75–90, 2009.

[30] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 193–204, 2010.

[31] W. Wu, L. E. Li, A. Panda, and S. Shenker. PRAN: Programmable radio access networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 6:1–6:7, 2014.

[32] Q. Yang, X. Li, H. Yao, J. Fang, K. Tan, W. Hu, J. Zhang, and Y. Zhang. Bigstation: Enabling scalable real-time signal processingin large mu-mimo systems. In *SIGCOMM*, 2013.

[33] L. Zhao, M. Li, Y. Zaki, A. Timm-Giel, and C. Gorg. Lte virtualization: From theoretical gain to practical solution. In *Proc 23rd ITC*, 2011.

[34] Q. Zheng, Y. Chen, R. Dreslinski, C. Chakrabarti, A. Anastasopoulos, S. Mahlke, and T. Mudge. Wibench: An open source kernel suite for benchmarking wireless systems. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 123–132, Sept 2013.

[35] Q. Zheng, Y. Chen, H. Lee, R. Dreslinski, C. Chakrabarti, A. Anastasopoulos, S. Mahlke, and T. Mudge. Using graphics processing units in an lte base station. *Journal of Signal Processing Systems*, 78(1):35–47, 2015.

[36] Z. Zhu, P. Gupta, Q. Wang, S. Kalyanaraman, Y. Lin, H. Franke, and S. Sarangi. Virtual base station pool: Towards a wireless network cloud for radio access networks. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011.