Towards Energy Efficient 5G vRAN Servers

Anuj Kalia   *Nikita Lazarev* [†]   *Leyang Xue* [‡]   *Xenofon Foukas*   *Bozidar Radunovic*   *Francis Yan*

Microsoft   [†]*MIT*   [‡]*University of Edinburgh*

Abstract

We study the problem of improving energy efficiency in virtualized radio access network (vRAN) servers, focusing on CPUs. Two distinct characteristics of vRAN software—strict real-time sub-millisecond deadlines and its proprietary black-box nature—preclude the use of existing general-purpose CPU energy management techniques. This paper presents RENC, a system that saves energy by adjusting CPU frequency in response to sub-second variations in cellular workloads, using the following techniques. First, despite large fluctuations in vRAN CPU load at sub-ms timescales, RENC establishes safe low-load intervals, e.g., by coupling Media Access Control (MAC) layer rate limiting with CPU frequency changes. This prevents high traffic during low-power operation, which would otherwise cause deadline misses. Second, we design techniques to compute CPU frequencies that are safe for these low-load intervals, achieved by measuring the slack in vRAN threads' deadlines using Linux eBPF hooks, or minor binary rewriting of the vRAN software. Third, we demonstrate the need to handle CPU load spikes triggered by control operations, such as new users attaching to the network. Our evaluation in a state-of-the-art vRAN testbed shows that our techniques reduces a vRAN server's CPU power consumption by up to 45% (29% server-wide).

1 Introduction

Virtualized RANs, which run the cellular radio stack on commodity servers instead of specialized hardware, are gaining adoption in modern cellular networks (e.g., 5G), owing to advantages such as a multi-vendor ecosystem, easier maintenance, and faster feature upgrades. This paper targets the energy efficiency of vRANs, focusing on the "Distributed Unit" (vDU) servers that perform real-time base station functions. Base stations are a major energy consumer: For instance, a recent report from China Mobile [49] estimates the deployment of 2.1 million 5G base stations (roughly three sectors/cells per base station) in China alone. Given today's 240-watt vRAN servers (§3.2), even a modest 1% reduction in server energy usage can translate to ~44 million kWh/year.

Two unique attributes of vRAN software—stringent real-time deadlines and black-box nature—preclude the use of standard CPU energy saving techniques. The lower vRAN layers running in the DU have hard deadlines on the order of the sub-ms Transmission Time Interval (TTI) of the wireless protocol, which must be met for functional correctness. Conventional energy-saving techniques, e.g., OS-controlled CPU frequency scaling or deep CPU sleep states, operate at tens of milliseconds and fail to meet these deadlines, with consequences as extreme as crashes or malfunctioning of the vRAN software. As a result, today's vRAN deployments disable these optimizations and instead maintain the CPU at a consistently high frequency regardless of the traffic [45].

This paper presents RENC, an energy-saving system that dynamically adjusts the DU's CPU frequency in response to sub-second changes in cellular workloads. Our design is based on the observation that cellular networks experience low utilization during most of their operational time. This leads to significant intervals of low CPU load where there is slack in the deadlines, i.e., a large deadline fraction remains unused, and there is room to reduce CPU frequency.

The central challenge for CPU energy saving in vRANs is the high variability in CPU load at sub-ms timescales. A sequence of TTIs that need few CPU cycles may be followed by a TTI with the maximum load, e.g., due to a traffic burst or a control plane operation. If the system is in low-power mode at this time, the DU can miss its deadlines with severe consequences (Section 2.1). The central idea in RENC is to establish "low-load" intervals for energy saving. We define "low-load" in terms of cellular traffic, as periods with data traffic below a small threshold (e.g., 10% or 1%), and with no expensive control plane operations. RENC needs to (1) separate these intervals from the opposite "high-load" intervals, and (2) determine safe low CPU frequencies for the low-load intervals. In high-load intervals, RENC keeps the CPU at a high frequency to ensure that deadlines are met.

For separation, RENC uses both proactive and reactive techniques, as well as careful timing. For example, we use the MAC scheduler for proactive rate limiting to prevent
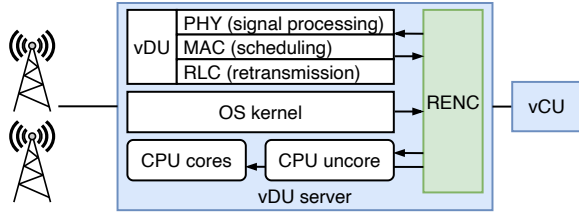
Figure 1: Overview of a vRAN deployment with RENC.

traffic bursts until RENC switches to high-load mode. RENC carefully orchestrates the timing of changes in MAC rate limits and CPU frequency, e.g., by ensuring that the increased CPU frequency is effective before lifting the rate limit. In addition to tackling user traffic-related CPU load, we also recognize control plane operations, such as user equipment (UE) attachments, as a cause of CPU load spikes. These can be handled in a reactive manner due to the longer control-plane latencies. RENC detects control plane messages using a small amount of telemetry from the vRAN and temporarily boosts CPU frequency to handle them.

The next challenge is to determine safe low CPU frequencies for the low-load intervals. This requires first measuring the "deadline slack" for each core, defined as the minimum unused fraction of the deadline. This is challenging because vRAN software is typically proprietary and closed-source, delivered to cellular operators by specialized vendors as a black-box binary. It is difficult for operators to directly add slack measurement code, especially since this must be done separately for each vendor. Instead, RENC aims to measure slack transparently to the extent possible: For interrupt-driven threads, we use eBPF [32] programmability in the Linux kernel scheduler, with no source code knowledge. For polling-based threads that do not yield to the OS, we use binary rewriting for a few DU functions. When the slack measured during low-load intervals remains high (e.g., above 10%) over a large number of samples, RENC chooses an iteratively lower CPU frequency for these intervals.

Inter-operability of RAN applications (such as energy-savings [6, Sec. 4.21]) across vRAN implementations is a key tenet of RAN virtualization and the "O-RAN" standards [4], which most vRAN vendors are adopting. The principle is to rely on interfaces exposed by vRAN components instead of internal details, to make wide deployment feasible. For this reason, we design RENC to run outside the vRAN software, and interact with it via well-defined interfaces.

Our evaluation in a commercial-grade 5G vRAN testbed, with two 5G 100 MHz 4x4 cells and nine commercial 5G UEs, shows that during periods of low traffic, RENC reduces the power consumption of vRAN CPUs and servers by up to 45% and 29%, respectively.

2 Background

Figure 1 shows a high-level overview of a vRAN deployment with RENC, focusing on the DU server that our work targets.

It consists of one or more (typically three, corresponding to a three-sectored cell site) radio units (RUs) that connect to the DU via an Ethernet fronthaul link. For sub-ms latency, the DU server is close to the cell site, in a cabinet or a small nearby datacenter. Multiple (e.g., 100s) DU servers connect to a "centralized unit" (CU) server, which handles the non-realtime, less compute-intensive layers of the vRAN stack.

The DU runs the lower layers of the vRAN stack: the Physical (PHY) layer that does wireless signal processing converting signal samples to bits and vice-versa, the Media Access Control (MAC) layer that schedules wireless resources, and the Radio Link Control (RLC) layer that handles reliable in-order delivery etc. RENC runs externally to the DU as a userspace agent, with an in-kernel eBPF component.

Sources of energy consumption Cellular networks consume a huge amount of energy, of which the RAN consumes a majority (e.g., GSMA estimates 73% [40]). Within the RAN, the RU and the DU consume the most energy. The energy split between the two is variable, ranging from near-equal for small radios, to 9:1 for large massive MIMO radios [31,49]. For example, our DU server consumes around 236 W, whereas each of our 4x4 indoor radios (one server can theoretically handle six such radios) consumes around 45 W when active. This works focuses on the DU server, whose energy efficiency is a topic of wide industry interest [37, 39, 42, 43, 54, 58, 62]. RU energy is complementary to our work, which we plan to tackle in the future. RUs require different types of optimizations, e.g., achieving large savings requires turning off analog components (e.g., via MIMO- and cell-sleep [24]), which are controlled at longer timescales (hours) than the sub-second variations that our work targets.

2.1 Distinct characteristics of the vRAN DU

Real-time deadlines. Unlike typical applications—which are the focus of most prior work on energy efficiency (cf. survey [50])—where work can be queued-up if CPU power is insufficient, the vRAN DU has strict real-time deadlines. The deadline for most threads is the TTI, which is 500 μs in 5G deployments in sub-6 GHz bands, which are vRAN's focus today; it is even lower (125 μs) in mmWave deployments. The computation needed per TTI is highly variable, and can change from near-zero to maximum in consecutive TTIs. The consequences of violating deadlines range from alarming (e.g., dropped calls), to catastrophic (e.g., a cell going offline due to the vRAN software crashing). This requires that energy management be done without affecting the vRAN's real-time performance.

Black-box nature. Although energy efficiency for real-time systems is well-studied [21, 57], the vRAN software's proprietary and closed-source nature makes it difficult to simply apply existing techniques. The basic method from this literature is to measure, at a given CPU speed, the task's deadline *slack*, i.e., the minimum unused fraction of the realtime dead-

line. The CPU speed can be reduced while the slack stays above a threshold. For example, if tasks use only 100 µs of a 500 µs deadline in the worst case (i.e., 80% slack), CPU speed can be safely reduced by a small amount, and the slack re-evaluated. While the high-level method is simple, applying it to vRANs presents several challenges (§4.1).

2.2 Techniques in traditional base stations

In traditional base stations (e.g., using in-house DSPs [13]), energy efficiency has been extensively studied for over a decade (cf. surveys [63, 64]). The problem is less explored for vRANs, which are a recent advancement. Non-virtualized base stations save energy by opportunistically turning off different hardware components. Debavilli et al. [29] provide a power model for different base station sleep modes, characterized by the time to enter and exit the mode, ranging from sub-100 µs to one second, and the power consumption in the mode. Base stations based on specialized hardware may have proprietary and tightly hardware-integrated sleep mode implementations that are co-designed for RAN processing [22, 24]. In contrast CPUs have only general-purpose sleep modes, which are not easily applicable to vRANs (§3.2).

2.3 CPU and OS features

CPU energy management consists of mainly (1) dynamic frequency scaling (called "Performance states", or P-states), and (2) CPU sleep states (called C-states). Today's vRANs typically disable these features (e.g., see Intel's documentation for FlexRAN [45]), consuming high power at all times [27]. RENC uses P-states and shallow sleep states (§3.2).

eBPF in the kernel. RENC uses eBPF to transparently measure deadline slack for some vRAN threads. eBPF [32] is a modern framework for running user-provided programs called codelets at predefined code points (called *hooks*) in unmodified binaries. eBPF codelets are written in a restricted C-like language, and are compiled to a bytecode that is verified for safety and then JIT-compiled to native code for high performance. The typical use case for eBPF is to augment the Linux kernel with new functionality. The recent Janus system [36] uses vendor-provided eBPF hooks in vRAN binaries to add custom functionality to the vRAN.

3 Motivating observations

3.1 Low utilization in cellular networks

Cell sites have low utilization, as a natural consequence of the physical cell tower infrastructure being dimensioned for peak traffic separately in each area. Today's vRAN servers also run at a CPU capacity sufficient to handle the maximum traffic at all times, frequently wasting energy.

Although large-scale traces of cell site traffic at sub-second granularity are not publicly available today, several studies provide evidence for this. For example, an Ericsson report on RAN energy efficiency for LTE concludes that *"even a*
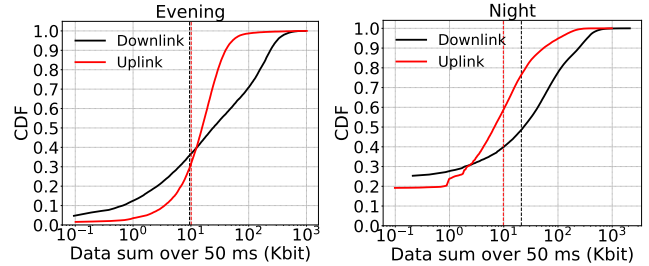

Figure 2: CDF of cumulative transport block size (TBS) measured every 50 ms, for two LTE cells. Vertical lines show 1% of peak traffic.

	C1	C1E	C6	P-states
Residency time	1 µs	4 µs	600 µs	N/A
Wakeup time	1 µs	4 µs	170 µs	N/A
Applicable to vRANs	✓	✓	?	✓

Table 1: Comparison of C-/P-states for our Ice Lake CPUs

future network, where hotspot peak-hour traffic might be 1,000 times greater than today, will still feature low traffic loads most of the time and in most locations." [56]. Another report from Ericsson mentions that *"Half of the sites have long periods of just idling and waiting for users to make use of the capacity available."* [24]. In fact, a study of RU energy efficiency by SK Telecom and NTT Docomo evaluates the power draw of only idle RUs, since that is the common case [31].

To illustrate this, we collected TTI-level traces from two commercial LTE cells, using the Falcon LTE sniffer [34]. Since RENC waits for 50 ms to transition to low-load (§4), we plot the CDF of the total traffic measured every 50 ms in Figure 2. The left figure shows a 9:30–9:47pm trace for cell #1, located in a major university; the right for 10:11pm–0:11am in cell #2, located in a residential area close to a different university. These are busy locations, and we expect that these evening/night measurements approximate typical cells during day hours. In cell #1, over 50% of the 50 ms intervals have less than 1% of the peak traffic; such periods reach 60–80% in cell #2. These intervals represent opportunities for RENC to save energy. We currently target durations with <1% traffic, but relaxing this to 10% is possible in our system (§9.1).

3.2 CPU power management

Choosing between P- and C-states. RENC uses P-states and shallow C-states, but not the deep sleep states. Modern Intel server processors support C1, C1E, and C6 states. C1 and C1E are shallow and save far less energy than C6, but their sub-5 µs latencies make them easily applicable to vRANs.

Can RENC use the deep C6 state? Table 1 shows why this is difficult. A C-state's "residency time" is defined as the minimum time that the core must stay in that state to save energy, to break-even from the sleep and wakeup overheads. For our Ice Lake CPUs, C6's high 600 µs residency time—required to flush the large L2 cache—plus its 170 µs wakeup time, exceeds the sub-500 µs TTI used in 5G networks. Most vRAN threads
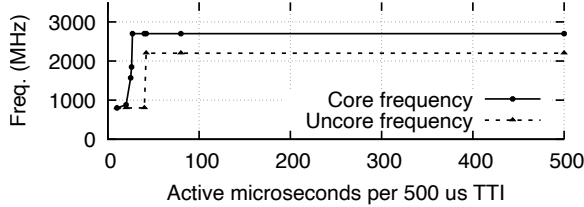
Figure 3: HWP-picked frequency for varying TTI activity levels

Component	Power
CPU (Xeon 6338N)	129 W
DIMMs (8x 32 GB)	14 W
Miscelaneous	93 W
Total	236 W

(a) Server power breakdown.
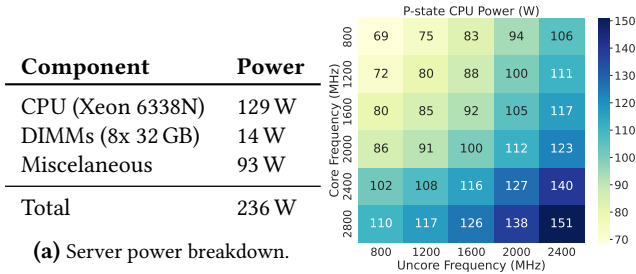
(b) Effect of core & uncore frequency

Figure 4: Components of vRAN server and CPU power

must run in every TTI—this pattern is followed by all six DUs that we have studied, including open-source [9, 15, 30], and proprietary [16, 33, 46]—so the CPU cores cannot remain inactive in C6 for long enough to save energy. We therefore focus on P-states instead, which allow the CPU to be active at all times and fine-grained frequency control. Recent advances reduce the impact of CPU frequency changes on core blocking time [28], and jitter on other cores [44]. C6 may be feasible for some DU threads with multi-TTI deadlines; we discuss this and its limitations in Appendix A.

CPU firmware controlled P-states. Can we simply let the CPU scale its own frequency? This does not work because the CPU's general-purpose P-state control loop is (1) too slow, and (2) not vRAN-aware. In modern CPUs, firmware-based frequency control (e.g., Intel's Hardware P-states [3], or HWP), is preferred over the older OS-based control. The firmware's simple 10 ms+ control loops are effective for general workloads, but vRANs require control at 1 ms timescales. For core P-states, we measured that HWP takes 60 ms to react after a sudden spike in CPU utilization; Schone et al. show that the uncore's reaction time is ~10 ms [59].

Figure 3 shows the CPU core and uncore frequency chosen by our server's HWP, (§ 8) in an experiment where one CPU core is active for varying fractions of a 500 μs TTI, and is asleep for the rest. We find that with only HWP picks the maximum core and uncore frequency with just 27 μs and 42 μs of activity, respectively. In contrast, RENC can recognize this thread as having high slack (over 90% of the TTI is unused), and choose a lower frequency.

CPU power breakdown. RENC focuses on CPU power since it is the largest component of DU server power. Figure 4a shows a wattage breakdown of our commercial-grade vRAN server (used in real vRAN deployments), when run-
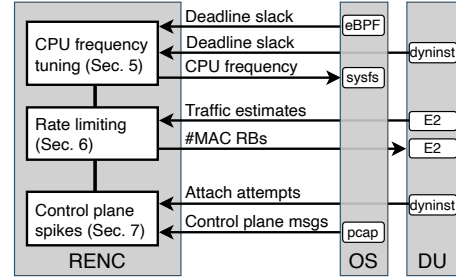

Figure 5: Overview of RENC's operation

ning two cells with no traffic and no energy management features enabled (details in §8). The CPU consumes the largest share of the power, eclipsing the power consumption of typically power-hungry components like DRAM. Our server's Baseboard Management Controller (BMC) not provide a breakdown of the "Miscellaneous" component, which divides among the motherboard, devices (NIC, PHY accelerator, NVMe SSDs), cooling fans, and conduction loss [17]. This remains fairly constant in our experiments regardless of user traffic, so we ignore it.

Uncore frequency. In addition to CPU cores, the CPU's "uncore"—the L3 cache, memory controller, PCIe controller, etc—is a major power consumer, since it represents a large fraction of the CPU's die area (e.g., over 50% [26]). Figure 4b shows the CPU's power draw at different core and uncore frequencies. Both are significant, so RENC tunes both.

4 Design overview

We target three design goals for RENC.

G-1. Minimize energy during low-load periods. Since most cell sites have low utilization most of the time (Section 3.1) RENC targets saving energy during these periods. We leave energy saving during high load to future work.

G-2. Transparency to vRAN software and platform. Since vRAN software is often proprietary and multi-vendor, and provided by a different vendor than the hardware/OS vendor, we design RENC following O-RAN [4] principles of clear interfaces that require only minimal information from the vRAN and platform. Specifically, we need only (1) names and deadlines of realtime DU threads, (2) the signatures of key functions in poll-mode vRAN threads, and (2) low-latency access to standard MAC layer metrics and control knobs.

G-3. Low resource overhead. RENC must consume few CPU resources, else it will negate some energy savings. In particular, RENC must not require a dedicated CPU core, which would reduce cores available for the DU.

4.1 Overview and challenges

C-1. Finding slack. When measured simply across all time, there may be no slack in some DU threads. Here we explain this intuitively, and show it experimentally in §9. Figure 6 shows three TTIs (#1, #2, and #n) for a DU thread with a
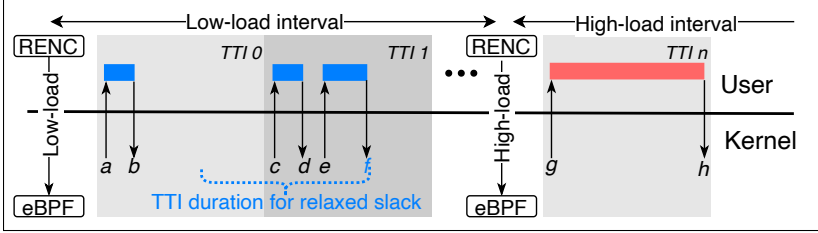
Figure 6: In this timeline for an interrupt-driven DU thread, RENC's eBPF program updates the relaxed slack for low-load intervals (blue) by summing over the dotted TTI-duration window at time f, i.e., $TTI - ((f - e) + (d - c))$. The userspace RENC agent tells the eBPF program the current low/high load type, so the latter ignores $TTI\ n$ with zero slack.

one-TTI deadline. TTIs #1 and #2 belong to RENC-controlled low-load intervals, by their low traffic rate and no expensive control plane operations. For these TTIs, the DU finishes work well before the deadline. In contrast, processing in the high-load TTI #n takes almost the entire deadline duration.

The key observation is that if we compute slack across all TTIs #1–#n, we will find no slack, i.e., the minimum unused deadline fraction is near-zero because of TTI #n. Without slack, RENC cannot reduce CPU frequency without risking deadlines, especially uncore frequency which affects all cores. However, if we carefully separate the low- and high-load intervals, we may find significant slack in the former. RENC introduces new techniques to (1) establish low-load intervals, (2) measure slack for precisely these intervals, and (3) limit energy saving to them. For low-load intervals, RENC iteratively computes the lowest safe frequency for each CPU core and the uncore. For high-load intervals, RENC runs the CPU at the highest configured frequency.

C-2. Separating low and high CPU load intervals. How can RENC ensure that the DU does not admit more load than the CPU can handle at low frequency? Prior work partially tackles this problem, e.g., PR3 [25], vrAIn [20] and CRT [55] use MAC-layer rate limiting to adjust traffic to match CPU frequency. However, these are best-effort, with no deadline guarantees that are crucial in commercial vRANs. For example, unlike less-performant open-source vRANs used in prior work [9, 15], our PHY (Intel's FlexRAN) is optimized to keep intermediate computation buffers sufficient for only the deadline period, minimizing its cache footprint for efficiency. Upon missing a deadline, there is no option but to crash the program. There are recent proposals that aim to alleviate this problem [38], but these require re-designing the DU.

Our solution is to couple MAC rate limiting with CPU frequency changes, by strictly ordering them. Before reducing frequency, RENC first applies a MAC layer rate limit to disallow sudden traffic bursts (§6). Similarly, RENC lifts the rate limit only after fully applying a CPU frequency increase. RENC also transitions to the high CPU load state on detecting CPU-intensive control-plane messages (§7).

C-3. Mostly-transparent slack measurement. Cellular operators typically receive the DU as a binary from a vendor, making it challenging to measure slack. We aim to infer slack externally as much as possible, for the two reasons. First, external measurement works generally for all DU implementations, increasing RENC's applicability to the diverse vRAN ecosystem. Second, some cores may run multiple realtime DU threads and have interference from unavoidable OS threads, in which case slack is best measured externally.

We have created two new techniques to measure slack, which rely on minimal information from the vendor that is easily available. First, we observe that the OS provides a vantage point to observe the activity of DU threads that yield to OS in every deadline interval. In the DU that we use (Section 8), five out of the seven types of realtime DU threads follow this pattern. For these threads, RENC measures slack with no code information from the vendor except thread names and deadline durations. Our insight is to use eBPF hooks in the Linux scheduler for this, since standard OS mechanisms are too coarse (§5).

For DU threads that do not yield to the OS, we require the function names and signatures of the realtime DU functions. Usually there is one top-level function that encapsulates the DU's computation in a deadline interval. RENC then uses binary rewriting to capture the start and end of these functions to measure the per-thread slack.

5 Measuring deadline slack

This section describes how RENC measures the slack for low-load intervals at runtime, while excluding high CPU load intervals whose slack can be zero; the next two sections describe how RENC keeps low-load intervals free from high load. RENC then uses slack measurements to iteratively reduce uncore and CPU cores' frequency until slack drops below a threshold (default 10% in our implementation) (§5.4).

In contrast to static measurement in prior work, RENC measures slack and tunes CPU frequency at runtime. For example, CRT [55] requires manual experiments to compute safe frequencies for different traffic loads by noting the CPU frequency at visible errors occur, e.g., UE disconnections and errors in the vRAN code. RENC provides an automated and principled solution to this problem. The heterogeneity of vRANs further complicates static approaches: vRAN sites may run different RAN configurations (e.g., different number of cells or wireless configuration parameters) with different server hardware, which change the CPU requirements.

5.1 Types of vRAN threads

RENC requires some basic information about the DU software from the RAN vendor, including whether the thread is interrupt-driven or busy-polling, and the thread's dead-

line. Table 6 shows our DU' threads, their types, and their deadlines. For busy-polling threads, RENC also requires the names and signatures of the task functions (typically one per thread) executed by the thread. In our DU, we use three functions corresponding to three PHY layer thread types: a function called at the start and end of each TTI; a function that handles fronthaul Ethernet packets; and a function for processing accelerator and shared-memory queues.

We develop two different approaches to measure slack.

Interrupt-driven threads are woken up by the OS at the start of each TTI, e.g., via a timer interrupt, or a semaphore from another thread. The thread performs its tasks for the TTI (e.g., MAC scheduling) and then yields. All MAC- and RLC-layer in our DU fall in this category. RENC uses eBPF hooks in the OS scheduler to measure their slack (§5.2).

Busy-polling threads run completely in userspace, i.e., they do not yield to the OS scheduler and instead constantly poll for work. In our DU, the PHY layer threads fall in this category, likely because of the PHY's high compute and efficiency requirements. We use binary rewriting for a few functions to measure the slack for these threads (§5.3).

5.2 Handling interrupt-driven threads

Can we simply use standard OS mechanisms to measure slack for these threads? This is not possible today, since the OS-exposed time measurements are too coarse, using "jiffies" (10 ms) as the unit of time. Our insight here is that eBPF provides the required fine-grained kernel programmability. To our knowledge, RENC is the first system to use eBPF to track thread execution time for energy saving, and we believe that it is a promising direction for non-vRAN systems as well.

In-kernel eBPF programs consist of simple, provably-safe code that is JIT-compiled to native machine code. The program has access to in-memory maps to store data, data structures, and high-resolution timers. RENC uses the eBPF hook at the Linux scheduler's `sched_switch` function, which is called whenever the thread running on a CPU core changes. Using the example in Figure 6 (see the caption for a detailed explanation), `sched_switch` can capture all user-kernel transitions, indicated by the arrows labeled with timestamps a–h. The correct low-load slack for the case shown is $TTI - ((f - e) + (d - c))$, since (1) the activity during $TTI\ 2$ exceeds $TTI\ 1$'s activity $(= b - a)$, and (2) the higher activity during $TTI\ n$ $(= h - g)$ must be ignored because it is in a high-load interval. Note that in practice we can have tens of active intervals per TTI, e.g., caused by the vRAN software's structure, as well as unavoidable kernel tasks such as IRQ and read-copy-update (RCU) pre-empting the DU threads.

5.2.1 Dividing between eBPF and userspace

What part of the slack measurement should run in eBPF within the OS, and what part in userspace? In RENC, we chose to implement slack measurement fully in eBPF, without any userspace involvement. The userspace RENC agent periodically (once every five seconds by default) retrieves the slack measurements from the eBPF program, and uses them to tune CPU frequency. This section describes the rationale behind this design, and how our implementation works.

An intuitive approach that we implemented first is to use a minimal eBPF program that simply passes all scheduling events to the RENC agent in userspace. This was convenient because all logic stays in userspace, easing development and debugging. However, we abandoned this approach because we found that sending messages to userspace for every scheduling event is fraught with issues. The compute overhead is a clear one: For example, with 10 thread switches per TTI (500 µs) across 10 DU cores, there are 200K kernel-user messages per second. This required dedicating a CPU core for the RENC agent to constantly poll an eBPF ring buffer for new messages, else messages are dropped.

We noticed more insidious issues, too. The high amount of kernel processing for message handling can get deferred behind the high-priority DU threads, causing subsequent bursts of kernel activity that cause the DU threads to miss their deadlines. Sending messages to userspace from a scheduler hook can also cause an infinite loop of messages leading to a kernel hang, since the act of sending a message can cause further scheduling events to handle the message.

5.2.2 In-kernel slack measurement

To avoid the issues with heavy scheduler-to-userspace messaging, we instead measure slack in the kernel. Surprisingly, we found that today there is no easy way for eBPF programs to know when a TTI starts and ends: At the time of writing, Linux's eBPF does not support the system wall-clock (`CLOCK_REALTIME`), supporting only monotonic clocks (e.g., `CLOCK_MONOTONIC`). The wall clock is the DU's heartbeat: it is synchronized with the RUs via the Precision Time Protocol (PTP), and is used to time TTIs. A patch to add wall clocks to Linux eBPF was submitted in 2020, but abandoned in part due to concerns about the likelihood of programmers' incorrect use of a non-monotonic clock [53].

Relaxed slack. Due to eBPF's lack of a TTI-synchronized clock, we use a conservative estimate of slack that does not rely on TTI boundaries, defined as follows (assuming a one-TTI deadline). We measure the maximum fraction in *any* TTI-length interval that the core is active in; the relaxed slack is one minus this fraction. The chosen interval may overlap TTIs (see Figure 6), making the relaxed slack a conservative estimate. This works well in practice because DU tasks run at the beginning of each TTI. For low-load intervals, the latter portion of TTIs is idle, so the selected interval will align with a TTI unless utilization is very high, in which case RENC does not attempt to save energy for this core.

To keep our implementation within the eBPF verifier's limits, we use a fixed-size array of intervals for each CPU core. Each interval is a pair of timestamps for when any thread was active, i.e., the kernel's idle task was not run-

Thread	Target function purpose and name
PHY signal processing	Called at start/end of TTI processing Name: `ebbu_pool_event_set_state()`
Fronthaul packet I/O	Per-symbol duration callbacks Name: `sym_ota_cb()`

Table 2: DU functions used for binary rewriting

ning. We deliberately chose to include *all* threads and not just DU threads, to include OS thread overheads which can be significant in real systems, e.g., following a NIC driver update that causes interrupts to land on DU cores. When a thread switch occurs at time t, the eBPF program sums all intervals overlapping with $(t - TTI, t)$. If the sum exceeds the current maximum, the program updates the maximum deadline utilization and the minimum slack.

The userspace RENC agent reads the per-core slack values from the kernel only when tuning frequency, i.e., once every five seconds by default. The user-kernel message overhead is now negligible, a significant reduction from the per-scheduling-event rate. We manually ensure that the in-kernel eBPF codelet has few instructions, and measure its overhead as <5% of the context switch cost (§9).

Handling load type changes. Our eBPF program maintains two arrays of intervals per core, one for each load type. When the userspace RENC agent decides to change the load type (§6), it indicates this to the eBPF program using a syscall to update a BPF map entry. The vertical "low-load" and "high-load" arrows between RENC and eBPF in Figure 6 show these updates. On every `sched_switch` invocation, our eBPF program sums up the intervals in the array corresponding to the current load type, keeping the two separate. While measurements for high-load intervals are not necessary for RENC, we still keep them to aid debugging.

5.3 Handling busy-polling threads

We use the popular `dyninst` framework to instrument the call and return instructions of the main datapath functions of busy-polling DU threads. Additionally, we use a framework (anonymized for double-blind submission) that allows inserting userspace eBPF programs at the instrumented code points. Since these programs can access the wall clock, slack measurement is straightforward.

In our DU, we use this approach for the busy-polling PHY layer threads. Table 2 lists the threads and the instrumented functions. For the PHY signal processing thread, we also need the description for the enum values of its argument, e.g., specifying whether TTI processing is starting or ending. We chose to not instrument one PHY thread type ("PHY SHM rings" in Table 6) because it splits its work across four functions, making it cumbersome to instrument all of them though this can be done if needed. This thread's processing is largely independent of the traffic load, so we manually confirm that it can safely run at the lowest frequency.

5.4 Iterative CPU frequency tuning

Collecting slack samples. At each CPU frequency configuration, RENC collects slack measurement samples for low-load intervals, filtering out high-load intervals. We continue collecting samples until the cumulative duration spent in low-load intervals exceeds a large value V, to try capturing the full range of CPU activity possible in low-load intervals. We expect V to be several hours in real deployments, but use a shorter 5-second interval for testing to speed convergence.

Prioritizing uncore frequency. After RENC has sufficient slack samples, it tries to reduce CPU frequency. We set our slack threshold to 10%, leaving some headroom for cases not covered by the observation period. We prioritize the uncore, since it has a large impact on power draw (§9.4) and affects all cores. If all cores have >10% slack, and uncore frequency is not at the minimum, RENC reduces it by the supported quantum (100 MHz). Otherwise, sys starts reducing CPU core frequencies for cores with >10% slack. Then RENC starts a new observation period.

6 Coupled CPU frequency and MAC control

This section describes how RENC coordinates changes in CPU frequency and MAC rate limits to separate low- and high-load intervals. Neither take effect immediately, so RENC must control their order and timing.

6.1 Forecasting DU traffic

RENC uses DU-exposed telemetry for uplink and downlink traffic to measure the current traffic demand, and determine when to transition to low or high load. Among the several metrics that can be used to measure traffic, we use the one that provides the earliest indication of traffic demand. This allows quickly removing the rate limit when there is a surge in traffic, minimizing the impact on RAN performance.

RENC requires low-latency access to DU telemetry and control interfaces that DU vendors typically already implement, e.g., for the O-RAN E2 interface [18]. Our DU vendor provides access to these via a local UDP port.

Uplink traffic estimation. We use the per-UE Buffer Status Reports (BSRs) to estimate the uplink load. UEs send BSRs to the DU to indicate buffered data size. These are sent *before* the UEs send the actual data, and hence provide an early indication of the uplink traffic demand.

Downlink traffic estimation. For downlink traffic, we use the throughput at the DU's interface with the CU. While this is not as early an indication as the BSRs, it is the earliest indication of downlink traffic that is available to the DU.

6.2 Low/high-load classification

Our current implementation simply uses recent traffic samples as indicators of future load, though more sophisticated techniques (e.g., time-series analysis) may be applied. We define $R_{dir,t}$ as the traffic sample received at time t, summed

over all UEs and cells, for uplink or downlink direction *dir*. We also define M_{dir} as the statically-measured maximum traffic sample for each direction; in our DU, this is 150 kB for the cumulative BSR, and 450 Mbps for the downlink throughput.

RENC transitions to low-load if all samples in the last n milliseconds are below 1% of M_{dir}; $n = 50$ in our implementation, i.e., we transition to low-load if there is little traffic for roughly three RTTs on our network (Figure 2). RENC transitions to high-load when the latest traffic sample exceeds 1% of M_{dir}. By not waiting for multiple >1% samples, RENC minimizes its impact on user-visible performance, at the cost of possibly reduced energy savings, e.g., when the high traffic sample is a transient spike. Note that some impact is acceptable, e.g., Ericsson uses a "Low Energy Scheduler" (LESS) that packs data in the frequency domain to make it sparse in the time domain [23].

6.3 Rate-limiting

RENC controls the number of wireless resource blocks (RBs, each RB corresponds to a small piece of spectrum) allocatable by the MAC for rate limiting, achieved via 3GPP's slice control interface [11]; this resembles the "ECO-Mode" optimization in some radios, which can power-down amplifiers for some RBs during times of low traffic [23]. This basic approach can be improved by using more information such as signal-to-noise (SNR) ratio, which affects the number of bits per RB. RENC allows a small percentage of RBs (10% by default in our implementation) to be used during low-load intervals. This is larger than the 1% threshold for traffic metrics, to account for the variable bits per RB.

6.4 Ordering and timing of changes

The process of changing CPU core/uncore frequency, or applying a MAC rate limit, is not instantaneous; RENC must ensure that these changes are applied in the correct order.

1. **CPU frequency and MAC rate limiting:** RENC must prevent high traffic while the CPU is at a low frequency.

2. **OS eBPF and MAC rate limiting.** RENC's kernel eBPF program must not pollute low-load slack measurements with those of high-load intervals (§5.2.2).

We next discuss how we apply these changes, and their maximum latencies measured when the system is running at RENC's low CPU frequency (§8.1). We use Linux's `sysfs` and MSR (Model-Specific Register) interfaces to control CPU core and uncore frequencies, respectively. We measure the maximum delays for these as 1100 μs and 2300 μs, respectively, which match prior work [59]. We define the time to change CPU frequency as the higher of the two, i.e., d_{cpu} = 2300 μs. Our DU applies RB changes in the TTI after it receives the request; The RB change latency equals RENC's maximum's latency to the DU's control UDP port (600 μs), plus one TTI (i.e., d_{rb} = 1100 μs). We use a syscall to update
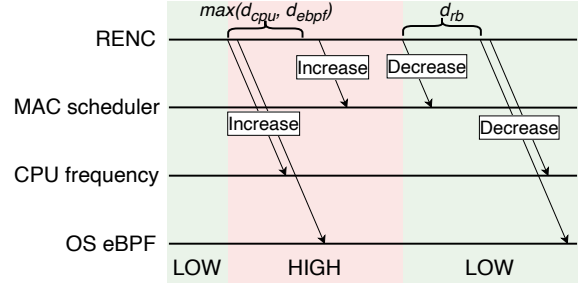

Figure 7: Ordering & timing of RENC's load-type transitions

Time	Event
34.448 s	Random access attempt from PHY (trigger)
34.469 s	CPU load spike; sudden slack decrease to 0%
48.515 s	UE Context Release Request from CU (trigger)
49.167 s	CPU load spike; sudden slack decrease to 68%

Table 3: Example of high load due to control messages

the load type in a BPF map in the eBPF program. This is a blocking call that takes at most 15 μs (= d_{ebpf}).

Figure 7 shows how RENC orders these changes. When transitioning from low to high load, it first increases CPU frequency and sets the eBPF load type to high-load. It waits for $max(d_{cpu}, d_{ebpf})$ for both changes to take effect, then lifts the MAC rate limit. Note that this sequence allows our kernel eBPF program to account slack for low-load TTIs to the high-load type, which is safe since it does not pollute the low-load slack measurements that RENC cares about. To transition to low load, RENC first applies the MAC rate limit, and waits for d_{rb} for this to take effect. It then reduces the CPU frequency and sets the eBPF load type to low-load.

7 Control operation CPU spikes

The discussion until now has focused on handling variations in the DU threads' slack caused by changes in the traffic load. However, this alone may not ensure that the DU meets its deadlines: In addition to the user traffic, the DU's control plane processing (e.g., UE attachments) can also be a source of sudden bursts of compute load. The key challenge here is detecting these spikes and reacting to them in a timely manner, with few modifications to the vRAN software for transparency and ease of deployment.

7.1 Discovering spike-causing events

The following insight helps to systematically and transparently determine the causes of compute spikes related to control messages. DU control messages are carried on formally-specified interfaces, i.e., the Functional API (FAPI) between the MAC and PHY [1], and the F1AP interface between the CU and DU [12]. Control messages can therefore be observed by simply interposing on the interface, ignoring the internal complex details of the vRAN implementation. The messages can be captured along with sudden dips in the DU threads'

slack (corresponding to a compute spike), and correlated to identify the causing message.

Table 3 shows example timeline for two sudden dips in a DU thread's slack when running a cell initially with no UEs attached, whose slack is initially 84%. These are caused by a (1) UE attaching and (2) detaching in our testbed (§8). The first dip is caused by a PHY-to-MAC Random Access Channel (RACH) Indication message, sent when a UE tries to attach. When this happens, the DU must initialize UE state, which causes the compute spike. The second spike is similarly caused by a F1AP UE Context Release Request message, which is sent by the CU to the DU when a UE detaches. Currently, we correlate events to spikes manually, but this can be automated using time series correlation methods. We expect that other events, such as UE handovers will also cause spikes, and leave this for future work.

7.2 Reacting to control messages at runtime

Fortunately, due to the relatively delay-tolerant nature of control messages, we can handle them reactively, i.e., a proactive approach such as RENC's rate limiting of data traffic is not necessary. For example, in Table 3, the attachment-related spike happens 21 ms after the random access control message. The detachment related dip happens 435 ms after the corresponding control message.

On detecting a spike trigger, RENC transitions to the high-load state, and stays there for the maximum time between the trigger and the corresponding spike. To leave sufficient error margins, currently RENC stays in the high-load state for 200 ms after the attachment trigger, and for 1 second for the detachment trigger. This is a conservative choice that can be optimized. In our DU, UE detachments force high-load for a long time which is undesirable, but these are rare since UEs typically detach infrequently to minimize signaling, staying in an idle/inactive state [41] during periods of inactivity.

To detect FAPI RACH messages, we intercept the PHY's top-level call to its FAPI interface to the MAC, which uses a publicly-available shared library ("Wireless Service Library", or WLS [5]). This library has a public, well-defined interface for compatibility with different MAC layer vendors [2], allowing us to de-parse the messages list to detect RACH Indications. To detect spike-causing F1AP messages, we use Linux's packet capture (pcap) interface and decode them to find the target message types.

8 Evaluation

We implemented RENC in 2500 source lines of code in C++ and C, and 100 lines of eBPF (`libbpf`) code.

8.1 Experiment setup

Table 4 summarizes our testbeds, which use state-of-the-art commercial-grade vRAN hardware and software that closely resembles real-world deployments [7]. To focus on the DU server, we run the CU and a commercial 5G core

DU Server	HPE DL110 Gen10 "telco" server Xeon 6338N CPU (32c), 8 ×32 GB DDR4 DIMMs 1 TB NVMe SSD, Intel E810 NIC, ACC100 card [10]
Software	Realime Linux v6.1; Intel FlexRAN PHY [46] CapGemini DU and CU, commercial 5G core
5G cells	FoxConn 4×4 RUs at 3.5 GHz, 100 MHz bandwidth 30 KHz subcarrier spacing, TDD (DDDSU)
Lab #1	Two cells in building, nine (4 + 5) Raspberry Pi UEs 13 realtime DU CPU cores, FlexRAN version 22.11
Lab #2	One cell, one OnePlus 5G phone and Pi UE 11 realtime DU CPU cores, FlexRAN version 22.03

Table 4: Evaluation testbeds' hardware and software

network on other servers. We use two labs, with different DU versions to test our design's generality. Lab #1 has two 5G cells on different building floors, with nine total Raspberry Pi UEs (four or five per floor) with Quectel RM500Q modems. We use the smaller lab #2 for experiments that need higher throughput, since it has better signal quality and modem. Our two-cell configuration is close to real-world deployments that often handle the sectors of a site. Our cells can achieve 30 Mbps uplink and 500 Mbps downlink TCP throughput, matching commercial 5G sub-6 GHz etworks [52, Fig. 6,7].

DU CPU core configuration. Our DU uses 13 realtime CPU cores for the two-cell configuration, and 11 for the one-cell configuration. Table 6 shows the latter; the two-cell configuration doubles the MAC layer cores. We did not design these configurations for RENC, but rather used the default stable configurations created over months with the vendors' help. The functional purpose of each thread is not important for RENC, but we include it for completeness. We minimize the energy effect of the unused DU server cores by setting them to the lowest 800 MHz frequency. Our default stable configuration disables Hyper Threading (HT), but RENC can be extended to support it by using the minimum of the two HT siblings' slack to tune the core's frequency.

Energy measurement. Our vRAN server's management interface (HPE iLO) provides full-server power measurements at coarse 10-second intervals. For fine-grained CPU power measurements, we use the CPU's RAPL (Running Average Power Limit) counters, which are accurate [48].

CPU frequencies. Our default server configuration before RENC used 3400 MHz for CPU cores, and 2400 MHz for the uncore, which we use as the baseline. RENC successfully separates-out low-load intervals that can run at the minimum CPU frequency that we allowed: 1000 MHz for all DU cores, and 800 MHz for the uncore. (Going further to 800 MHz for the CPU cores saves very little energy but costs 20% CPU performance.) We report per-core slack measurements, and the marginal impact of core and uncore frequencies in §9.

8.2 Idle-mode power consumption

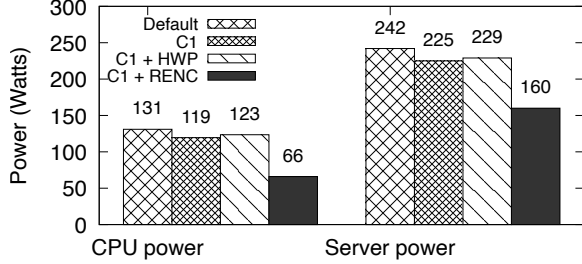We start by measuring RENC's idle-mode power (§3.1) savings, which is a key energy metric for RAN equipment [31].

Figure 8: Idle-mode CPU and server-wide power consumption, with different energy saving configurations.

	DL Mbps	UL Mbps	Ping ms
Without RENC	486–520	29.6–29.7	27.1 (3.3)
With RENC	499–520	29.7–29.7	27.9 (3.4)

Table 5: Basic Internet performance without and with RENC

We attach all nine UEs in lab #1, to the two cells, which then remain idle. We use three baselines: (1) the default configuration with no energy optimizations, (2) enabling C1 states, and (3) CPU's hardware-based automatic frequency scaling (HWP, §3.2). When using HWP, we set its "energy-performance bias" policy in the BIOS to favor energy. We ignore C1E states since they save negligible energy (< 1%) in our experiments. Figure 8 shows that that enabling C1 states reduces CPU power from 131 W to 119 W, and server power from 242 W to 225 W. Enabling RENC on top of C1 further reduces CPU power by 45% to 66 W, and server power by 29% to 160 W, respectively.

Comparison with Intel Hardware P-states. CPU power draw with HWP enabled (123 W) is 1.8x higher than with RENC. In fact, enabling HWP *increases* power draw, because HWP does not allow any software to control the frequency of unused cores, which HWP runs at frequencies higher than the 800 MHz that we use without HWP. In addition, HWP runs all PHY cores and most MAC and RLC cores at too-high frequencies (2.5–2.7 GHz) (c.f. Figure 3). Third, it picks a too-high uncore frequency (2200 MHz); artificially forcing the uncore (permitted via Model Status Registers even when HWP is active) to 800 MHz brings down CPU power to 83 W, which is still higher than RENC's 66 W by 26%.

8.3 Energy savings with data traffic

Effect on network performance. RENC aims to save energy without significantly affecting network performance. Here we study high-level metrics, and run micro-benchmarks in §9. Since performance with multiple UEs is noisy due to spectrum sharing and interaction with TCP, we use lab #2 in this experiment. We use SpeedTest [8], and ensure that all tests use the same SpeedTest server. We run three tests both with and without RENC, and report the maximum and minimum values. Since SpeedTest's ping latency varies widely, we use a different ping application to measure the average and standard deviation of latency to cloudflare.com. Table 5
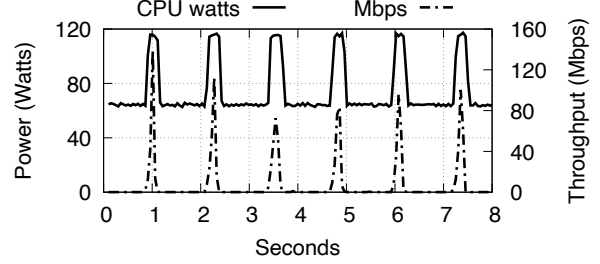

Figure 9: One UE repeatedly downloading a 1 MB file.
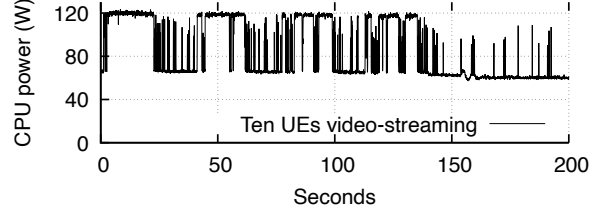

Figure 10: Energy saving with nine video-streaming UEs.

shows the results. The throughputs and latency of the two configurations are similar, which meets our goal.

We next evaluate RENC in the larger lab #1 with real traffic.

File downloads. In this experiment, one Pi UE repeatedly downloads a 1 MB file from a lab server at 1 s intervals. Figure 9 shows the DU CPU's power draw measured at 10 ms intervals, along with downlink bandwidth. RENC successfully reacts to the load spikes by increasing CPU frequency, and reduces it when the load drops.

Video streaming. We use video streaming as an example application to show how RENC can save energy even during active user sessions. Here we use all nine Pi UEs, which stream a 720p video from a popular Internet site, using a headless mpv player [51]. The UEs start at roughly the same time. Figure 10 that, in the beginning, RENC keeps the DU server primarily in high-load mode as the UEs fetch the initial parts of the video. After the UEs buffer the video, there are gaps between video chunk downloads, during which RENC successfully saves energy. The average power draw with RENC is 83 W, compared to 121 W without (not shown).

Traffic mix. We use the nine UEs to create a traffic mix representative of a real-world workload: we run three video streams, two 1 MB file downloads and two 256 kB uploads at 1 s intervals, two long-lived 1 Mbps iperf3 download streams, and one 1 Mbps iperf3 upload stream. Figure 11 shows how
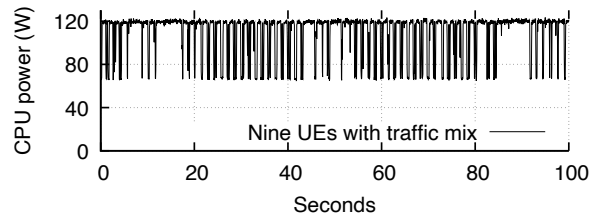

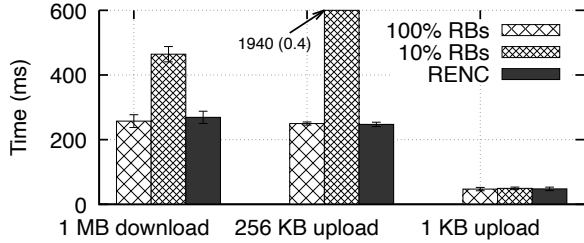**Figure 11:** Energy saving with nine UEs running a traffic mix.

Figure 12: File transfer time with MAC rate limiting.

Figure 13: Handling of control message spikes.

RENC saves energy during the traffic gaps, reducing average CPU power from 121 W to 109 W.

9 Microbenchmarks

We next evaluate some aspects of RENC's design in isolation.

9.1 Effect of MAC rate limiting

To quantify how RENC's MAC rate limiting affects network performance, we measure the time for small file transfers that stress RENC's reactiveness; our SpeedTest experiment showed RENC's negligible impact on long transfers. Since RENC dynamically scales the allowed RBs between 10% and 100%, we show the download time with static 100% and 10% RB limits for reference. The experiment uses a simple Python HTTP client and server. Since downlink speed is higher than uplink, we use a larger 1 MB for downloads than uploads (256 kB). These are smaller than average mobile web page sizes today (e.g., 2 MB in one study [14]). Figure 12 shows that transfer time with RENC is close to the static 100% allocation (<2% higher), and better than the static 10% allocation. For small 1 kB uploads, common in IoT devices [61], all three perform similarly since the DU always stays under 10% RBs.

Sensitivity analysis. We have focused on a 1% threshold for traffic metrics, but this is tunable. Increasing it to 10% allows RENC to save more energy, but may affect user performance unless the algorithms in §6 are improved. For example, using a 10% BSR threshold in the 256 kB upload experiment increases the mean transfer time from 200 ms to 231 ms, due to the DU's slower reaction to the uplink traffic burst.

RU power. RENC's rate limiting may slightly increase the amount of time that the RU is active, increasing its power draw. We currently lack the precise RU power measurements needed to quantify this, but we expect it to be small since RENC lifts its rate limit in milliseconds in response to traffic.

9.2 Slack measurement

We conduct the following experiment to test RENC's establishment of low-load intervals that are safe for CPU frequency scaling. Since this experiment requires stressing the DU's compute resources, we use lab #2 to generate high traffic. We run two tests: without RENC, and with RENC, shown in the rightmost two columns of Table 6. In each test, we attach the phone, run two SpeedTest sessions, and then detach the phone. In the first case, the DU runs without
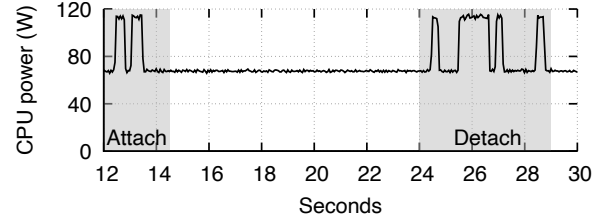
RENC at a constant low frequency of 1000 MHz for all cores and 800 MHz for the uncore; this configuration frequently crashes the DU due to deadline violations, but we run it for this experiment. In the second case, RENC is enabled.

We make three observations (Table 6). First, without RENC, several DU threads have zero slack. The MAC scheduler thread has 8% slack, but that leaves insufficient safety margins. This confirms that this DU cannot run at this low frequency without RENC, since slack measured simply across all time is zero. Second, RENC successfully establishes low-load intervals with sufficient slack for frequency scaling. Third, some threads (e.g., the PHY threads and the RLC timer thread) have significant slack even without RENC, making it possible to statically configure them to run at a low frequency. ("PHY timer" has lower slack with RENC because its processing time depends more on CPU frequency than user traffic.)

Importantly, note that knowing whether a CPU frequency configuration is safe, or which threads have high slack, is difficult without our new slack measurement techniques. For example, simply running coarse-grained tool like `top` always shows <9% and <27% or the MAC-to-PHY and MAC scheduler threads, respectively, but their actual TTI deadline utilization without RENC reaches 100% and 92%, respectively.

9.3 Control message handling

Figure 13 shows the DU's CPU power draw when a Raspberry Pi UE attaches and detaches in lab #2. There are two high-load intervals during attachment because of a failed attempt followed by a successful one, which is common in cellular networks. The detachment event has one long high-load interval, and multiple three short ones caused by spurious attachment attempts that our UE makes when we turn on flight mode to detach it. This experiment shows how RENC successfully handles CPU load spikes from control processing, and handles unexpected behaviors of UEs. RENC also handles the more challenging case with nine UEs simultaneously attaching and then detaching.

9.4 Breakdown of energy savings

Table 7 shows that both uncore and CPU core frequency scaling contribute significantly to energy savings. Reducing the uncore frequency alone from 2400 MHz to 800 MHz reduces CPU power by 32% from 117 W to 80 W. Adding core frequency scaling reduces CPU power by a further 16% from 80 W to 67 W. A takeaway from our work is that uncore frequency scaling provides a convenient and effective knob for

Thread name	Description	Running mode	Deadline	Slack w/o RENC	Low-load slack with RENC
F1 worker	Communication with CU	Interrupt-driven	$1 \times$ TTI	0%	64%
RLC timer	RLC timekeeping	Interrupt-driven	$1 \times$ TTI	62%	75%
RLC worker	RLC processing	Interrupt-driven	$1 \times$ TTI	0%	76%
MAC scheduler	Wireless scheduler	Interrupt-driven	$1 \times$ TTI	8%	79%
MAC-to-PHY	MAC-PHY messaging; UE context management	Interrupt-driven	$1 \times$ TTI	0%	72%
PHY BBU pool (4x)	Wireless signal processing	Busy-polling	$3 \times$ TTI	20%	28%
PHY timer	Symbol timing; callbacks	Busy-polling	Symbol $(\frac{TTI}{14})$	72%	56%
PHY SHM rings	Network/accelerator I/O	Busy-polling	$1 \times$ TTI	N/A (§5.3)	N/A (§5.3)

Table 6: Threads with realtime priority running at the DU: low CPU frequency vs RENC.

CPU cores freq	Uncore freq	CPU power
3400 MHz	2400 MHz	117 watts
3400 MHz	800 MHz	80 watts
1000 MHz	2400 MHz	105 watts
1000 MHz	800 MHz	67 watts

Table 7: Effect of CPU core (12x) and uncore frequency scaling

reducing CPU power. Since it is just one value, an operator can adopt RENC by gradually reducing uncore frequency while monitoring low-load slack.

9.5 Overhead of RENC

Userspace agent. We measure the CPU cost of RENC's userspace agent as 20% of a CPU core, when running the CPU at low frequency. RENC does not require a dedicated CPU core nor does it busy-poll. It runs on shared cores used for other tasks such as the DU's management/logging threads.

Kernel eBPF. We measure the execution time of RENC's eBPF program as 70 ns per invocation at 1 GHz, using Linux's BPF stats functionality via the `bpftool` command. This is tiny when compared to the total cost of context-switching between threads, which is around 3 μs on our system even when running at 3.4 GHz. The eBPF program is called twice per context switch, costing <4.6% of a context-switch.

10 Related work

Section 2.2 covered energy saving techniques in traditional base stations. We present related work in other areas below.

Approaches that target vRANs. CRT [55] statically pre-computes CPU core frequencies for different MAC modulation and coding schemes (MCS); this is done in an adhoc way, e.g., by checking if errors appear in code execution. Compared to CRT, RENC adds (1) principled deadline slack measurement which is done at runtime, (2) coupling of MAC scheduling and CPU frequency changes for safety, (3) handling of CPU spikes from control messages, and (4) evaluation in a real testbed. Ayala-Romero et al. [19] develop Bayesian algorithms to jointly optimize vRAN power consumption and performance. However, the system does not account for realtime deadlines. While these systems target

open-source RAN implementations (OpenAirInterface and srsRAN), RENC tackles the additional challenges that arise in closed-source, binary-only setups, as well as in the comparatively higher complexity of commercial vRAN sofware.

Multi-server approaches. China Mobile has published details of a large RAN energy efficiency effort in China, which uses machine learning to pick cells to power-off [49]. A large body of theoretical work studies the bin-packing of RUs to the fewest DU servers, aiming to save energy during low load periods [60]. Cellular operators also optimize energy consumption at longer time horizons to match the diurnal traffic patterns [49], e.g., MIMO-level sleep, and cell-level sleep [22, 24]. RENC's single-server and sub-second optimization is complementary to these approaches.

Single-server approaches. Recent techniques for sharing the DU's realtime CPU cores with non-realtime workloads may be adapted for DU energy saving: Concordia [35]'s prediction of PHY task execution time can be used to place PHY cores in a deep C6 sleep state to save energy, although it will need to overcome C6's high residency time (Table 1, Appendix A). Nuberu [38] re-designs the PHY layer to better co-exist with interference from other workloads. If a DU implements a Nuberu-like technique, it can simplify RENC design by allowing rare deadline violations.

11 Conclusion

RENC takes the first step towards building a complete energy saving system for servers running commercial vRAN software. The difficulty lies in the closely-guarded nature of the software code that makes it a black box for operators, coupled with sub-ms real-time deadlines that must be met for functional correctness, as well as a highly bursty CPU load. We solve this challenge by carefully establishing intervals of low load that are safe for energy saving, and using new systems techniques such as our eBPF program to measure the available slack for CPU frequency reduction. An evaluation in a state-of-the-art vRAN testbed shows promising results, with up to 45% reduction in CPU energy compared to default settings.

A Appendix

Applicability of C6 sleep state to vDUs

C6 may work for some DU threads with multi-TTI deadlines. In our DU, these are only FlexRAN's PHY signal processing threads (which run on four of 11 cores in our DU, Section 8), whose deadline is three TTIs. Although FlexRAN provides a partial implementation for C6 [43], it is incomplete at the time of writing; FlexRAN's public C6 description suggests that it puts threads to sleep one TTI at a time with usleep(500) [43], which will not save energy on our CPUs, and may have been designed for older processors e.g., Cascade Lake, with 276 µs [47] C6 residency time.

On our Ice Lake CPUs, we found that a CPU core goes into C6 when an application requests a sleep of at least 650 µs (e.g., via usleep(650)). The call to usleep(650) returns after up to 820 µs, due to the 170 µs wakeup latency of the C6 state.

References

[1] 5G FAPI: PHY API specification. https://www.smallcellforum.org/reports/5g-fapi-phy-api-specification.

[2] FlexRAN 5G New Radio Reference Solution L1-L2 API Specification. https://docs.o-ran-sc.org/projects/o-ran-sc-o-du-phy/en/latest/fapi_5g_tm_overview.html#reference-documents.

[3] Intel CPU frequency scaling drivers. https://wiki.archlinux.org/title/CPU_frequency_scaling.

[4] O-RAN Alliance: Operator Defined Open and Intelligent Radio Access Networks. https://www.o-ran.org/.

[5] O-RAN Software Community: Wireless Service Library. https://docs.o-ran-sc.org/projects/o-ran-sc-o-du-phy/en/latest/wls-lib.html.

[6] O-RAN Use Cases Detailed Specification 11.0. https://www.o-ran.org/specifications.

[7] Rakuten Symphony Symware Phase Two Begins with Plans to Commercially Deploy 30,000 Units in Japan. https://symphony.rakuten.com/newsroom/rakuten-symphony-symware-phase-two-begins.

[8] Speedtest by ookla - the global broadband speed test. https://www.speedtest.net/. Accessed: yyyy-mm-dd.

[9] SRS: Software Radio Systems. https://www.srs.io/.

[10] v4l2-loopback device. https://github.com/umlaeute/v4l2loopback.

[11] 5G Network Resource Model (NRM) (3GPP TS 28.541 version 16.6.0 Release 16). https://www.etsi.org/deliver/etsi_ts/128500_128599/128541/16.06.00_60/ts_128541v160600p.pdf, November 2020.

[12] F1 Application Protocol (F1AP) (3GPP TS 38.473 version 15.8.0 Release 15). https://www.etsi.org/deliver/etsi_ts/138400_138499/138473/15.08.00_60/ts_138473v150800p.pdf, January 2020.

[13] The case for integrated high-performance RAN processing. Ericsson Blog, 2020.

[14] The Growth of Web Page Size. https://www.keycdn.com/support/the-growth-of-web-page-size, Nov 2022.

[15] OpenAirInterface. https://gitlab.eurecom.fr/oai/openairinterface5g, March 2023.

[16] Radisys 5G NR Software Suite. https://www.radisys.com/connect/connectran/5g, n.d.

[17] Kazi Main Uddin Ahmed, Math HJ Bollen, and Manuel Alvarez. A review of data centers energy consumption and reliability modeling. IEEE Access, 9:152536–152563, 2021.

[18] ORAN Alliance. E2 service model (e2sm). O-RAN Fronthaul Working Group, O-RAN.WG3.E2SM-R003-v03.00, 2023.

[19] Jose A Ayala-Romero, Andres Garcia-Saavedra, Xavier Costa-Perez, and George Iosifidis. Orchestrating energy-efficient vRANs: Bayesian learning and experimental results. IEEE Transactions on Mobile Computing, 2021.

[20] Jose A. Ayala-Romero, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Albert Banchs, and Juan J. Alcaraz. VrAIn: A Deep Learning Approach Tailoring Computing and Radio Resources in Virtualized RANs. In The 25th Annual International Conference on Mobile Computing and Networking, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems: A survey. ACM Transactions on Embedded Computing Systems (TECS), 15(1):1–34, 2016.

[22] Thomas Berglund and Helen Huynh. Energy Efficiency of Radio Units and its Impact on RAN Energy Consumption. 2017. Masters Thesis in Electrical Engineering, Ericsson AB, Lund University, Faculty of Engineering.

[23] Olof Bjering and Lakshmi Prasad. An energy efficient radio base station. 2018. Faculty of Engineering, Lund University.

[24] Blomgren, Anton and Ornstein Mecklenburg, Kasper. Energy Optimization of Radio NGR Micro G1, 2016. Student Paper.

[25] Nishant Budhdev, Mun Choon Chan, and Tulika Mitra. PR3: Power Efficient and Low Latency Baseband Processing for LTE Femtocells. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 2357–2365, 2018.

[26] Hsiang-Yun Cheng, Jia Zhan, Jishen Zhao, Yuan Xie, Jack Sampson, and Mary Jane Irwin. Core vs. uncore: The heart of darkness. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[27] Private communication with a major vRAN vendor, 2024.

[28] Intel Corporation. Enhanced Power Management for Low-Latency Workloads. https://networkbuilders.intel.com/docs/networkbuilders/power-management-enhanced-power-management-for-low-latency-workloads-technology-guide-1617438252.pdf, 2023.

[29] Bjorn Debaillie, Claude Desset, and Filip Louagie. A Flexible and Future-Proof Power Model for Cellular Base Stations. In *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pages 1–7, 2015.

[30] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. *Agora: Real-Time Massive MIMO Baseband Processing in Software*, page 232–244. Association for Computing Machinery, New York, NY, USA, 2020.

[31] NTT DOCOMO. Green Mobile Network: Energy Saving Efforts by SK Telecom and NTT DOCOMO. https://www.docomo.ne.jp/english/binary/pdf/corporate/technology/rd/docomo6g/GreenMobileNetworksWhitePaper_22February2023.pdf, Feb 2023.

[32] eBPF.io. eBPF. https://ebpf.io/, March 2023.

[33] CapGemini Engineering. CapGemini 5G gNodeB. https://capgemini-engineering.com/nl/en/services/next-core/wireless-frameworks/, March 2023.

[34] Robert Falkenberg and Christian Wietfeld. FALCON: An accurate real-time monitor for client-based mobile network data analytics. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2019.

[35] Xenofon Foukas and Bozidar Radunovic. Concordia: teaching the 5G vRAN to share compute. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 580–596. ACM, 2021.

[36] Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, and Zhihua Lai. Taking 5G RAN Analytics and Control to a New Level. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, page to appear. Association for Computing Machinery, 2023.

[37] Fujitsu. Fujitsu launches sustainable 5G vRAN to deliver potential reductions in CO2 emissions of over 50 percent. https://www.fujitsu.com/global/about/resources/news/press-releases/2022/0224-01.html, 2022.

[38] Gines Garcia-Aviles, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Pablo Serrano, and Albert Banchs. Nuberu: Reliable RAN virtualization in shared platforms. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 749–761, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Srihari Das Sunkada Gopinath, Sandeep Burugupally, and Ajeet Singh Nathawat. An adaptive power management method for radio access network data plane systems. In *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1129–1134. IEEE, 2022.

[40] GSMA Intelligence. Going Green: Benchmarking the Energy Efficiency of Mobile, 2021.

[41] Sofonias Hailu, Mikko Saily, and Olav Tirkkonen. RRC state handling for 5G. *IEEE Communications Magazine*, 57(1):106–113, 2018.

[42] Red Hat. Red Hat and Arm Collaborate to Deliver More Energy-Efficient 5G and vRAN Solutions. https://www.redhat.com/en/blog/red-hat-and-arm-collaborate-deliver-more-energy-efficient-5g-and-vran-solutions, 2023.

[43] Intel. Intel Network Builder Insights Series: The Full vRAN Experience. https://networkbuilders.intel.com/university/webcasts/the-full-vran-experience, Sep 14, 2022 2022. Slides: https://www.brighttalk.com/resource/core/408572/the-full-vran-experience---deck_873458.pdf.

[44] Intel. Dynamic Frequency Scaling for Mixed Criticality Real-Time Scenarios on 11th Generation Intel® Core™ Processors. https://www.intel.com/content/www/us/en/content-details/723446/dynamic-frequency-scaling-for-mixed-criticality-real-time-scenarios-on-11th-generation-intel-core-processors.html, 2023.

[45] Intel. FlexRAN README. https://github.com/intel/FlexRAN/blob/08ddafedce6a1b5690221f881738bddaa7093588/README.md, Aug 2023.

[46] Intel. FlexRAN Reference Architecture for Wireless Access. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html, March 2023.

[47] Intel. Native hardware idle loop for modern Intel processors. https://github.com/torvalds/linux/blob/master/drivers/idle/intel_idle.c, n.d.

[48] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[49] Tong Li, Li Yu, Yibo Ma, Tong Duan, Wenzhen Huang, Yan Zhou, Depeng Jin, Yong Li, and Tao Jiang. Carbon emissions and sustainability of launching 5g mobile networks in china, 2023.

[50] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V. Vasilakos. Cloud Computing: Survey on Energy Efficiency. 2014.

[51] mpv.io. mpv: a free, open source, and cross-platform media player. https://mpv.io/, May 2024.

[52] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. A variegated look at 5G in the wild: performance, power, and QoE implications. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 610–625, 2021.

[53] Ashkan Nikravesh and Bimmy Pujari. spinics.net patch: bpf: Add realtime clock to BPF. https://www.spinics.net/lists/bpf/msg23699.html, July 2020.

[54] NTT. NTT Succeeded in Low Power Consumption of 5G Virtualized Radio Base Station (vRAN). https://group.ntt/en/newsrelease/2023/05/24/230524b.html, 2023.

[55] Ujjwal Pawar, Bheemarjuna Reddy Tamma, and Franklin A Antony. Traffic-Aware Compute Resource Tuning for Energy Efficient Cloud RANs. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 01–06, 2021.

[56] Ericsson Technology Review. Radio network energy performance: shifting focus from power to precision. https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/radio-network-energy-performance-shifting-focus-from-power-to-precision, 2014.

[57] Sonal Saha and Binoy Ravindran. An experimental evaluation of real-time DVFS scheduling algorithms. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[58] Samsung. Samsung Announces the Next Phase of Its 5G vRAN. https://news.samsung.com/global/samsung-announces-the-next-phase-of-its-5g-vran, 2023.

[59] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. In *2019 International Conference on High Performance Computing and Simulation (HPCS)*, pages 399–406, 2019.

[60] Tshiamo Sigwele, Atm S Alam, Prashant Pillai, and Yim F Hu. Energy-efficient cloud radio access networks by cloud based workload consolidation for 5G. *Journal of Network and Computer Applications*, 78:1–8, 2017.

[61] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[62] Vodafone. Vodafone, Wind River, Intel, Keysight Technologies and Radisys test green Open RAN. https://www.vodafone.com/news/technology/vodafone-wind-river-intel-keysight-technologies-radisys-test-green-open-ran, 2022.

[63] Jingjin Wu, Yujing Zhang, Moshe Zukerman, and Edward Kai-Ning Yung. Energy-efficient base-stations sleep-mode techniques in green cellular networks: A survey. *IEEE communications surveys & tutorials*, 17(2):803–826, 2015.

[64] Yi Zhang, Łukasz Budzisz, Michela Meo, Alberto Conte, Ivaylo Haratcherev, George Koutitas, Leandros Tassiulas, Marco Ajmone Marsan, and Sofie Lambert. An overview of energy-efficient base station management techniques. In *2013 24th Tyrrhenian International Workshop on Digital Communications-Green ICT (TIWDC)*, pages 1–6. IEEE, 2013.