Understanding On-device Bufferbloat For Cellular Upload

Yihua Guo[†], Feng Qian[♮], Qi Alfred Chen[†], Z. Morley Mao[†], Subhabrata Sen[‡]
[†]University of Michigan [♮]Indiana University [‡]AT&T Labs – Research
{yhguo,alfchen,zmao}@umich.edu fengqian@indiana.edu sen@research.att.com

ABSTRACT

Despite the extensive characterization of the growth of cellular network traffic, we observe two important trends not yet thoroughly investigated. First, fueled by the LTE technology and applications involving wearable devices and device-to-device (D2D) communication, device *upload* traffic is increasingly popular. Second, the multi-tasking and multi-window features of modern mobile devices allow many concurrent TCP connections, resulting in potentially complex interactions. Motivated by these new observations, we conduct to our knowledge the first comprehensive characterization of cellular upload traffic and investigate its interaction with other concurrent traffic. In particular, we reveal rather poor performance associated with applications running concurrently with cellular upload traffic, due to excessive on-device buffering (*i.e., on-device bufferbloat*). This leads to significant performance degradation on real mobile applications, *e.g.,* 66% of download throughput degradation and more than doubling of page load times. We further systematically study a wide range of solutions for mitigating on-device bufferbloat, and provide concrete recommendations by proposing a system called QCUT to control the firmware buffer occupancy from the OS kernel.

Keywords

Upload; Bufferbloat; Cellular Networks; Radio Firmware

1. INTRODUCTION

The explosive growth of mobile devices and cellular networks shows no sign of slowing down. We notice two important trends not well explored in previous work, namely *user-generated traffic* and *multi-tasking*.

IMC 2016, November 14-16, 2016, Santa Monica, CA, USA
© 2016 ACM. ISBN 978-1-4503-4526-2/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2987443.2987490

On one hand, the mobile traffic paradigm is undergoing a shift from being dominated by download to a mix of both download and upload, due to a wide range of emerging apps enabling user-generated traffic such as media content upload to social networks (*e.g.,* Facebook videos), background synchronization, cloud-based offloading, HD video chat, machine-to-machine (M2M), and device-to-device (D2D) communication, *etc.* The prevalence of upload is further fueled by increasingly ubiquitous access to LTE networks, which provides uplink bandwidth of up to 20Mbps.

On the other hand, the frequent use and rich function of mobile devices give rise to multi-tasking, which enables users to interact with multiple applications at the same time. Previously, due to the limitation of mobile operating system and the device processing power, older phones and Android systems only support a single application at foreground interacting with the user. Newer phones allow users to use multiple apps simultaneously (*a.k.a.* "multi-window"). Even without multi-tasking, a single foreground app can still trigger multiple concurrent TCP flows. A recent study [20] shows that around 28% of the time for each mobile user there are concurrent TCP flows.

Motivated by the above, in this paper, we conduct to our knowledge the first comprehensive, quantitative, and cross-layer measurement study of cellular *upload* traffic and its interaction with concurrent traffic, using combined approaches of analyzing large network traces and conducting controlled lab experiments. Our contributions consist of the following.

Characterization of upload traffic (§3). We characterized the cellular upload traffic by measuring its volume, duration, rate, concurrency, and impact on latency from a large network trace collected from an IRB-approved user study[1] involving 15 users for 33 months. We found that although the majority of today's smartphone traffic is still download, the upload traffic can be significant. Upload can last for up to 11 minutes with 226 MB of data transferred. In particular, the upload speed can achieve up to 12.8Mbps (median 2.2Mbps for 10MB+ flows) in today's LTE networks, fa-

[1]This study was conducted entirely with data collected from active and passive measurements at the University of Michigan and was approved by the University of Michigan IRB (Institutional Review Board) approval number HUM00111075.

cilitating many applications that upload rich user-generated traffic. We also found that large upload tends to have higher RTT, and upload traffic may also increase the RTT experienced by concurrent download.

An anatomy of on-device queuing for upload traffic (§4.1-§4.3). For cellular upload, we found a significant fraction of the end-to-end latency occurs on the end-host device instead of in the network, due to the large buffer inside the mobile device. Specifically, we made two key observations. First, contrary to the common understanding [41, 22] that *(i)* excessive queuing delay (*a.k.a.* "*bufferbloat*") happens mostly inside the network (or near-edge network elements), and *(ii)* cellular upload traffic is less likely to incur queuing delay due to its low data rate, our findings suggest that *bufferbloat for upload traffic can frequently occur on mobile devices accessing diverse types of networks (HSPA, LTE, and even Wi-Fi), across different devices.* On-device bufferbloat can cause significant latency increase up to 4 seconds, or 100x of the network RTT, on off-the-shelf Android devices. This implies that when an upload is in progress, *the on-device queuing delay in fact eclipses the network delay.*

Second, we identified the root cause of such excessive on-device queuing. It can happen at different layers including application buffer, OS TCP buffer, and the Queuing Discipline (Qdisc) in the OS kernel. In particular, we found excessive queuing also frequently happens at the *cellular firmware buffer*, whose occupancy can, for example, reach up to 300KB while accounting for 49% of the end-to-end delay when the uplink bandwidth is 2Mbps. More importantly, the cellular firmware buffer distinguishes itself from other on-device buffers in that *its occupancy plays a role in the cellular control plane which in turn affects base station's scheduling decision and achievable uplink throughput.*

Accurate achievable uplink throughput estimation (§4.4). The excessive buffering at various layers makes accurate estimation of achievable cellular uplink throughput challenging. We found that surprisingly, using the same algorithm, the throughput estimated at upper layers (TCP and Qdisc) deviates from the lower-layer throughput estimation by 136% and 70% on average. We proposed a method that accurately infers the uplink throughput by leveraging lower-layer information from cellular control-plane messages.

Quantifying the impact of uplink bufferbloat (§5). We illustrated that large upload traffic significantly affects the performance of concurrent TCP download, whose average RTT is increased by 91% and average throughput is reduced by 66%. We found *such severe performance degradation is predominantly caused by on-device buffers*, because the uplink ACK stream of TCP download shares the same Qdisc and cellular firmware buffers with concurrent upload, and is thus delayed mainly due to the on-device bufferbloat. We further quantitatively demonstrated concurrent upload incurs significant user experience degradation on real applications, including web browsing (219% to 607% increase of page load time with concurrent upload), video streaming (57% reduction of playback bitrate and frequent stalls), and VoIP.

Mitigating on-device bufferbloat (§6,§7). Due to the uniqueness of on-device bufferbloat, we found existing mitigation solutions, such as changing TCP congestion control, tuning TCP buffer size, reducing Qdisc sizes [7], prioritizing delay-sensitive traffic, and applying Active Queue Management (*e.g.,* CoDel [30] and PIE [34]) are not capable of effectively mitigating the excessive buffering. This is because *(i)* they cannot be directly implemented at the cellular device driver, and *(ii)* they are unaware of the interplay between the driver buffer and the cellular control plane. We therefore designed and implemented a new solution called QCUT to control the firmware buffer occupancy from the OS kernel. QCUT is general (independent of a particular driver implementation), lightweight, and effective. Our lab experiments show that QCUT effectively reduces the cellular firmware queuing delay by more than 96% while incurring little degradation of uplink throughput. We deploy QCUT on a user study involving 5 users for one week. The results indicate QCUT significantly improves the application QoE when concurrent upload is present.

Although we identified the on-device bufferbloat problem in today's HSPA/LTE networks, we anticipate it continues to affect future wireless technologies, whose uplink bandwidth will remain below the downlink bandwidth (*e.g.,* 50Mbps vs. 150Mbps for LTE Advanced [3]) causing the cellular uplink to often remain the end-to-end bottleneck link. More importantly, higher network speed and cheaper memory facilitate device vendors and cellular carriers to deploy larger buffers that exacerbate on-device (and in-network) bufferbloat.

Paper Organization. After describing the experimental methodology in §2, we conduct a measurement study of today's upload traffic in §3. We reveal the on-device queuing problem in §4, and quantify the impact of upload on mobile apps in §5. We then describe how QCUT mitigates on-device queuing in §6 and evaluate QCUT as well as existing mitigation strategies in §7. We discuss related work in §8 before concluding the paper in §9.

2. EXPERIMENTAL METHODOLOGY

To comprehensively study cellular upload traffic, we carried out controlled lab experiments (§2.1) and analyzed data collected from a user study with 15 participants (§2.2).

2.1 Controlled Local Experiments

We conduct controlled experiments using off-the-shelf smartphones and commercial cellular networks. Our devices consist of the following: *(i)* Samsung Galaxy S3 running Android 4.4.4 with Linux kernel version 3.4.104, using Carrier 1's LTE network[2], *(ii)* Samsung Galaxy S4 running Android 4.2.2 with Linux kernel version 3.4.0, using Carrier 1's LTE network, *(iii)* Samsung Galaxy S3 running Android 4.0.4 with Linux kernel version 3.0.8, with access to Carrier 2's 3G network and *(iv)* Nexus 5 running Android 6.0.1 with Linux kernel version 3.4, using Carrier 1's LTE network. We also set up dedicated servers located at the University of Michigan with 64-core 2.6GHz CPU, 128GB memory, 64-bit Ubuntu 14.04 OS for experiments. Both mobile phones

[2]We anonymized three large U.S. cellular carriers' names as Carrier 1, 2, and 3.

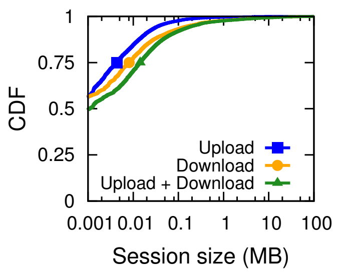


Figure 1: Traffic volume distribution of user sessions.

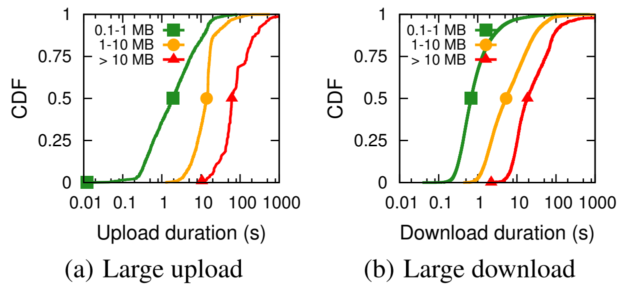


Figure 2: Flow duration distributions.

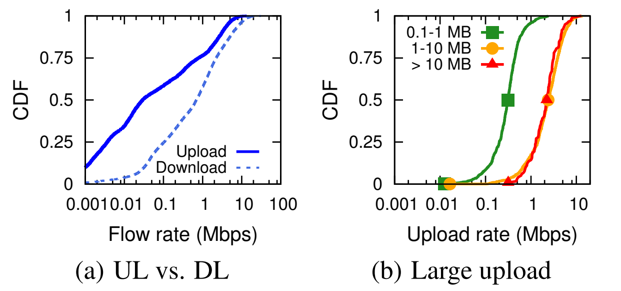


Figure 3: Flow rate distributions.

and the servers use TCP CUBIC [19], the default TCP variant for Linux/Android, unless otherwise mentioned. We conducted the experiments during off-peak hours. For each setting, we repeat the experiment for at least 10 times and report the average metrics unless otherwise noted.

For TCP throughput and RTT measurement, the mobile device first establishes a TCP connection to one of the dedicated servers. Then this TCP connection is used to transfer random data without interruption. For bidirectional data transfer (*i.e.,* simultaneous upload and download) experiments, the mobile device establishes two TCP connections to two servers, one for download and the other for upload to eliminate server-specific bottlenecks. To measure throughput, we ignore the first 10 seconds to skip the slow start period and calculate the throughput every 500ms from transferred data.

2.2 Network Traces from a User Study

We also leveraged network traces collected from an IRB-approved smartphone user study by distributing instrumented Samsung Galaxy S3 phones to 15 students. Each of the 15 participants was also given unlimited LTE data plan. The phones were instrumented with data collection software (with very low runtime overhead). It continuously runs in the background and collects full packet traces in `tcpdump` format including both headers and payload. We collected 900GB of data in total from January 2013 to October 2015. We used an idle timing gap of 1 minute to separate *user sessions* of the same device.

3. UPLOAD TRAFFIC MEASUREMENT

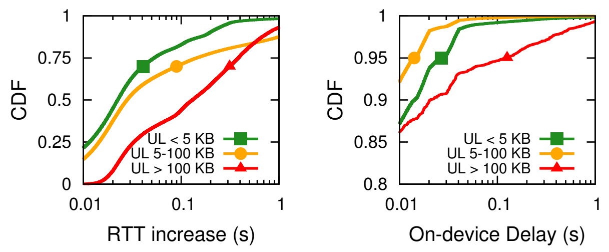
In this section, we perform a measurement study of upload traffic in today's cellular networks, using the traces collected from our user study (§2.2).

Traffic volume. Figure 1 plots the distributions of download, upload, and overall traffic volume of user sessions. We only consider TCP/UDP payload size when computing the session size. Today's mobile traffic is dominated by download, whose average size is about one order of magnitude larger than that of upload. About 2.2% of user sessions carry more than 1MB of downlink bytes, while only 0.4% upload more than 1MB data in the user study trace. We notice a major source of user-consumed traffic is video, which accounts for about half of download traffic.

However, we observed that the fraction of upload bytes is indeed non-trivial. Within the top 20% of user sessions (in terms of their overall transferred bytes), the 25th, 50th, and 75th percentiles of the fractions of upload traffic are 9%, 20%, and 42%. Across all user sessions, the corresponding fractions are higher, *i.e.,* 19%, 50%, and 57%. The upload of one user session even lasts for 11 minutes with 226 MB of data transferred. We expect that in the future, the fraction of upload traffic will keep increasing because of the increasingly popular user-generated traffic. Compared to smartphones, wearable and IoT devices may incur even more upload traffic, due to their ubiquitous sensing capabilities.

Flow duration. We use inter-packet arrival time to divide a TCP flow into multiple segments with a threshold of 1 second. We also use the threshold of 1 second to eliminate idle period of a TCP flow. We then divide these segments into *upload bursts* that only have TCP payload in uplink, and *download bursts* with only TCP downlink payload, based on the direction of transferred TCP payload. Given a TCP flow, we define its *upload duration* as the total duration of all uplink bursts. Similarly, the *download duration* is the total duration of all downlink bursts. Figure 2(a) plots the upload durations for flows with large upload traffic volume (100KB to 1MB, 1MB to 10MB, and at least 10MB). As expected, larger flows tend to be longer in duration. For flows with large download traffic volume, as shown in Figure 2(b), their download duration exhibits distributions qualitatively similar to those of upload duration, yet with the main difference being that the download duration is statistically shorter, largely due to the higher downlink bandwidth compared to the uplink bandwidth, as to be measured next.

Flow rate. We compute the upload (download) rate of



(a) RTT increase of upload (b) On-device delay of download

Figure 4: Delay distributions.

a TCP flow by dividing the total bytes of all upload (download) bursts by its upload (download) duration that is defined above. We only consider flows whose upload/download duration are longer than a threshold, which is empirically chosen to be 3 seconds, as the "rate" of a very short flow is not very meaningful. Figure 3(a) compares upload and download rates for the user study trace. Statistically, download is faster than upload, largely due to their differences in the underlying channel rates of the LTE radio access network. On the other hand, Figure 3(b) indicates larger upload flows (larger than 1MB) tend to have higher rates. In the user study dataset, for flows that upload 1 to 10 MB data, their 25%, 50%, and 75% percentiles of upload rates are about 1.4Mbps, 2.4Mbps, and 3.8Mbps, which are comparable or even higher than those of today's many residential broadband networks. For 10MB+ flows, the maximum achieved throughput are 12.8Mbps for the user study traces. Such high upload speed provides the infrastructural support for user-generated traffic.

Flow concurrency. We explore the concurrency of TCP flows per user. The result is shown in Figure 5. For every one-second slot in each user session, we count the number of TCP flows that are transferring data. For the user study trace, for 28.2% of the time (*i.e.,* 28.2% of the one-second slots across all user sessions), there exist at least two TCP connections that perform either upload or download. The results indicate that concurrent TCP transfers are quite common on today's mobile devices. Motivated by this, we study the interplay between uplink and downlink traffic, a previously under-explored type of concurrency, in §5.

RTT Dynamics. We next study the RTT dynamics of cellular upload from the user study trace. The RTT is measured by timing the timestamp difference between each uplink TCP data packet and its corresponding ACK packet captured by `tcpdump`. We then study the fluctuation of upload RTT using the following methodology. First, we split each user session into one-second slots and discard slots without uplink traffic. We also discard slots whose download traffic volume is non-trivial (using 5KB as a threshold). The purpose is to eliminate the impact of concurrent download on upload. Second, for each one-second slot s, we compute its *RTT increase* $I(s) = \text{mean}_p\{\text{RTT}(p) - \text{MinRTT}(p.flow)\}$ over all data packets $\{p\}$ within the slot. RTT(p) is the measured RTT, and MinRTT$(p.flow)$ is the minimum RTT of upload stream of the TCP flow that p belongs to. Third, we plot in Figure 4(a) the distributions of $I(s)$ across all slots grouped by their upload size. As shown, the upload RTT is indeed highly "inflatable", and larger upload tends to incur much higher RTT. This resembles the "bufferbloat" effect that is well studied for download [41, 22], and motivates us to conduct a comprehensive investigation of bufferbloat for cellular upload in §4.

Impact of Upload on Download Latency. We are also interested in how upload impacts download latency, which is quantified as follows. We first generate one-second slots using a similar way as employed in Figure 4(a), but this time we only keep slots with *both* upload and download traffic. Then for every slot, we compute the average on-device delay of download. Note that since our user study traces were collected on client devices, we are only able to measure the *on-device* component of the download RTT *i.e.,* t_1 shown in Figure 6(b). Next, in Figure 4(b), we plot the distributions of the on-device download delay grouped by the size of per-slot upload size as is also done in Figure 4(a). We clearly observe that concurrent upload affects the on-device download delay because the ACK stream (for download) and the data stream (for upload) share several on-device buffers. We will conduct an in-depth investigation on this in §4 and §5.

Summary. Overall, we found that although the majority of today's smartphone traffic remains to be download, the upload traffic can still be large. In particular, the median upload speed is 2.2Mbps for 10MB+ flows and can achieve up to 12.8Mbps in today's LTE networks, enabling many applications to upload rich user-generated traffic. We also found that large upload tends to have higher RTT, and upload traffic may also increase the RTT experienced by concurrent download. Furthermore, it is quite common that multiple TCP flows are transferring data concurrently on a mobile device, leading to complex interactions possibly among uplink and downlink flows to be investigated soon.

4. ON-DEVICE QUEUING DELAY OF UPLOAD TRAFFIC

We conduct a thorough analysis of the latency characteristics for cellular upload traffic. We found a significant fraction of the latency happens on the end-host device instead of in the network (§4.1). In particular, in §4.2, we discover the root cause of large Qdisc and firmware buffers playing major roles in causing the excessive on-device delay, whose prevalence across devices and carriers are shown in §4.3. We also found in §4.4 that on-device queuing may significantly impact accurate uplink throughput estimation.

4.1 Overall Delay Characterization

When a mobile device is uploading data, its packets will traverse various buffers in the protocol stack, as illustrated in Figure 6: TCP buffers, link-layer buffers (Linux queuing discipline), radio firmware buffers. Each buffer may incur queuing delay. As a result, we may get different RTT values if we conduct measurements at different layers. In this work, we focus on three RTT measurements defined bellow.

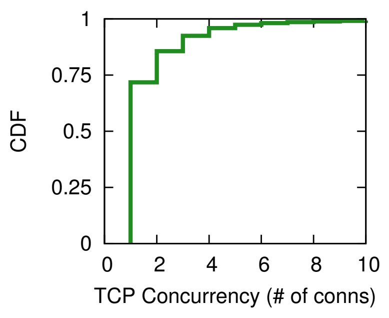


Figure 5: TCP concurrency distribution.

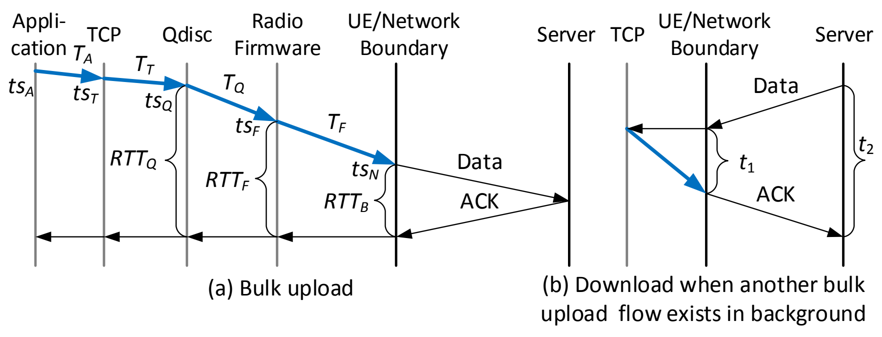


Figure 6: On-device bufferbloat (in thick blue).

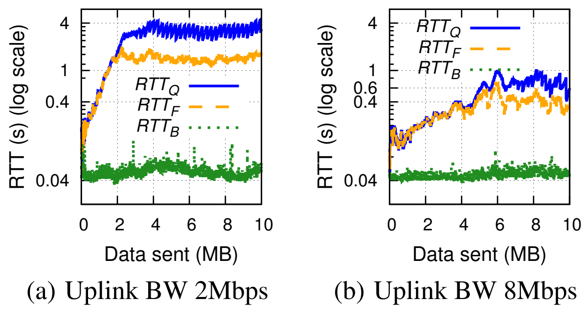


Figure 7: Overall latency characterization for a single TCP upload flow under two network conditions.

- RTT_B consists of only the delay a packet experiencing in the network. It does not include any delay caused by on-device buffer (*B* stands for "base").
- RTT_F includes RTT_B and the delay incurred by the buffer in the radio firmware, which usually resides on the cellular chipset of a mobile device (*F* stands for "firmware").
- RTT_Q includes RTT_F, plus the delay incurred by the queing discipline (Qdisc), the link-layer buffer in the main memory managed by the OS (*Q* stands for "Qdisc").

Similarly, we can also define RTTs measured at higher layers (TCP, application). Nevertheless, RTT_B, RTT_F, and RTT_Q are our particular interests, because their corresponding network path or buffers are *shared* by multiple applications. As we will show in §5, if upload and delay sensitive traffic coexist, the former may severely interfere with the latter due to the shared nature of lower-layer buffers. In contrast, the higher-layer buffers are usually not shared.

We now measure RTT_B, RTT_F, and RTT_Q by performing bulk data upload over TCP on Samsung Galaxy S3, using Carrier 1's LTE network. RTT_Q and RTT_F can be directly measured by `tcp_probe` [8] and `tcpdump`, respectively. RTT_B can only be indirectly estimated. For a given upload trace, we keep track of the buffer occupancy and enqueue/dequeue rates of the firmware buffer. We then use them to estimate the firmware queuing delay[3]. RTT_B is then calculated by subtracting RTT_F measured from `tcpdump` trace by the estimated queuing delay. The results of two representative experiments with different uplink bandwidth (2Mbps and 8Mbps, which are 50% and 98% percentiles of upload rates measured from Figure 3(b), respectively) are shown in Figure 7. Note the Y axes are in log scale.

As shown in both plots of Figure 7, at the beginning of the TCP upload, RTT_F increases steadily, and quickly outweighs RTT_B. When the uplink bandwidth is 2Mbps (8Mbps), after 2MB (4MB) of data has been sent out, RTT_F is increased to around 1.3s (330ms), which is much larger than RTT_B maintaining stably at around 50ms. Meanwhile, RTT_Q starts to exceed RTT_F, and becomes twice as large as RTT_F after another 2MB of data is uploaded. The absolute difference between RTT_Q and RTT_F is as high as 3s and 680ms in Figure 7(a) and 7(b), respectively.

Overall, we found that for cellular upload, surprisingly, the RTT observed by mobile devices' TCP stack (RTT_Q) can be significantly larger than the RTT perceived by `tcpdump` (RTT_F), which further far exceeds the pure network RTT (RTT_B). Depending on the uplink bandwidth, RTT_Q and RTT_F can be 22x$\sim$100x and 6x$\sim$24x of RTT_B, respectively, during the steady phase of TCP bulk upload. We call such a phenomenon *on-device bufferbloat* since it is caused by excessive queuing delay on the mobile device, as opposed to the network, which is regarded as the main source of excessive queuing for cellular downlink traffic [22]. As we will demonstrate later, on-device bufferbloat has deep implications on, for example, uplink bandwidth estimation, multi-tasking performance, uplink scheduling algorithms, and on-device buffer management.

4.2 Root Cause of On-device Queuing

We now explore the root cause of the excessive on-device queuing delay. We begin with an overview of how outgoing TCP packets on the sending path traverse the Linux kernel (also used by Android) and radio chipset. As shown in Figure 8, an application invokes the `send()` system call at time ts_A and the data is put into TCP buffer by kernel through TCP sockets at time ts_T. Note ts_T may be later than ts_A if the socket is blocked. In Linux, a packet is stored

[3]The detailed methodology of firmware buffer occupancy estimation is described in §6.2. Since the radio firmware we use only reports firmware buffer occupancy of up to 150KB, we validate our methodology when the occupancy is smaller than 150KB and then use it to infer the buffer occupancy at any time in the trace.

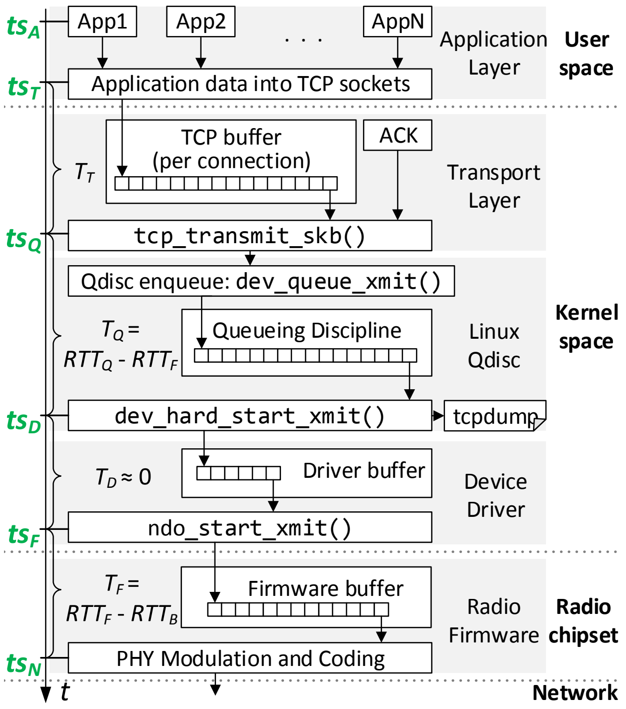


Figure 8: Packet processing and transmission on Android devices.

in a data structure called skb. A TCP packet is encapsulated into skb (with its TCP header being added) and sent to IP layer in `tcp_transmit_skb()` at time ts_Q. After being processed at the IP layer, the skb with TCP/IP header is subsequently injected to the queuing discipline (Qdisc) when `dev_queue_xmit()` is called. When the driver is ready to transmit more data, `dev_hard_start_xmit()` is called by the kernel to dequeue the packet from Qdisc to the driver buffer at time ts_D. Similarly, when the radio firmware of the chipset is ready to receive a packet from the driver, `ndo_start_xmit()` will be called to enqueue the packet to the firmware buffer at time ts_F. The kernel usually does not have direct control over the logic of radio firmware, which determines when to actually transmit the packet at time ts_N.

As mentioned in §4.1, in this study we focus on the queuing delay below the transport layer, as lower-layer buffers are usually shared, causing potential interference across multiple apps. We now describe lower-layer queuing in details.

In-kernel Queuing. To identify where on-device queuing occurs exactly in the kernel, we use Linux kernel debugging tool jprobe to log timestamps of the aforementioned function calls. We found that the queuing delay in the queuing discipline (Qdisc), denoted as T_Q, is almost identical to the difference between RTT_Q and RTT_F. Besides, the delay between `tcp_transmit_skb()` and `dev_queue_xmit()`, as well as the driver queuing delay (T_D) are negligible. This indicates that when sending out traffic in cellular networks, packet queuing in Qdisc dominates the on-device queuing delay in the kernel. Also, as another validation, we observe

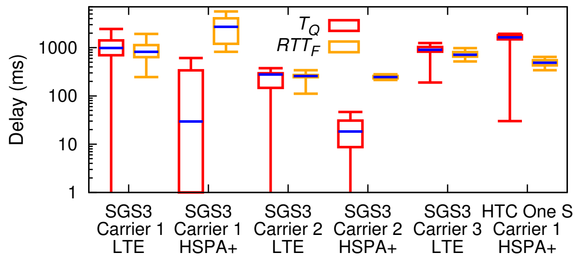


Figure 9: On-device queuing delay on diverse devices and cellular carriers.

a strong correlation (around 0.86) between the queuing delay and the amount of traffic in Qdisc.

Firmware Queuing. In LTE uplink, the data to be transmitted from applications is processed and queued in the RLC (Radio Link Control) buffer[4], which is physically located in the cellular chipset firmware. The amount of data available for transmission on the UE (*i.e.,* the firmware buffer occupancy) is provided to the eNodeB through control messages called *Buffer Status Reports* (BSR). BSR can report 64 levels of buffer size with each level representing a buffer size range [9]. The highest level of BSR is 150KB or above. Based on BSR from all UEs, the eNodeB uplink scheduler assigns network resources to each UE for uplink transmission in a centralized manner. The eNodeB sends control messages called *Scheduling Grants* to inform a UE of the scheduling decision. A UE is only allowed to transmit on the physical uplink shared channel (PUSCH) with a valid grant.

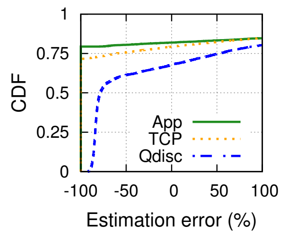
We observed that the BSR quickly increases to the highest level (150KB+) when there is large LTE upload traffic. Also there exists a strong correlation (around 0.73) between RTT_F and buffer level in BSR. Leveraging the BSR information, we measured that the actual firmware buffer occupancy can reach several hundreds of KBs (using a more accurate algorithm described in §6.2), and the firmware queuing delay (T_F) can reach 400ms with 8Mbps uplink. By subtracting the RTT_F by the firmware queuing delay, we can estimate RTT_B, which is constantly low (*e.g.,* around 50ms for Carrier 1), as shown in Figure 7.

Overall, the above findings have two important implications. First, cellular uplink scheduling is performed in a centralized manner, different from that in Wi-Fi networks where clients autonomously sense the wireless channel to transmit data and avoid collision in a distributed way. Second, the firmware buffer distinguishes itself from other on-device buffers in that its occupancy plays a role in the cellular control plane which in turn affects eNodeB's scheduling decisions and the achievable uplink throughput.

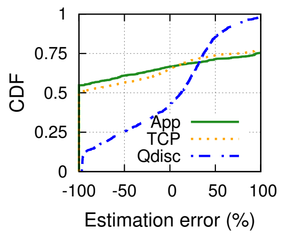
4.3 Prevalence across Carriers & Devices

We show the prevalence of on-device bufferbloat in Figure 9 by repeating the upload experiments on various net-

[4]In the remainder of this paper, we use the general term "firmware buffer" to refer to the RLC buffer.



(a) Measurement error with 20ms interval



(b) Measurement error with 100ms interval

Figure 10: Uplink throughput measurement error at different layers.

	Throughput		RTT	
	% AVG decrease	% RSD increase	% AVG increase	% RSD increase
SGS3 C1 LTE	66	253	91	37
SGS3 C1 HSPA+	8	25	7	9
SGS3 C2 LTE	10	36	10	20
SGS3 C3 LTE	80	192	86	42
HTC One S C1 HSPA+	22	260	10	91

Table 1: Impact of upload on download performance on different devices, vendors, and networks (C1, C2, and C3 refer to Carrier 1, 2, and 3 respectively).

works using two different devices. For each setting, we report the 5th, 25th, 50th, 75th, and 95th percentiles of T_Q and RTT_F. We observe on-device bufferbloat on all settings in LTE, with median T_Q larger than 200ms. Regarding the HSPA+ network, T_Q is small (around 20ms) for SGS3 using Carrier 2. This is because the TCP sending buffer size is configured to be small by Carrier 2 on this device (we will discuss the impact of TCP buffer size in §6.1). Yet across all settings, we found that the RTT_F is much larger than the estimated RTT_B, indicating that excessive firmware queuing happens on all devices and carriers.

4.4 Uplink Throughput Measurement

Often applications (*e.g.,* real-time multimedia apps) need to know the instantaneous network throughput. The lower-layer information provided by firmware enables accurate cellular throughput measurement. Recall in §4.2 that the UE can only send the amount of data up to the scheduling grant. If a portion of the grant is not used, the firmware uses padding to indicate the unused part. The padding size is also reported by the firmware. Therefore, by subtracting the scheduling grant by the padding size, we can calculate the amount of data sent out from the device, as well as the uplink throughput (the padding is not transmitted).

Since the above approach directly utilizes lower-layer information from the cellular control plane, it gives the ground truth of cellular uplink throughput. An interesting question is, compared to this ground truth, how accurate is the throughput measured at upper layers? We quantify this in Figure 10, which plots the measurement error at Qdisc, TCP, and application layer where we use a slide window of 100ms and 20ms to estimate uplink throughput during a bulk upload. The results indicate that the throughput estimation at higher layers are highly inaccurate, with the root mean square being 141%, 136%, and 70% at the application layer, the transport layer, and the Qdisc, respectively, when the estimation interval is 100ms. Reducing the interval further worsens the accuracy. The root cause of such inaccuracy is again the on-device bufferbloat: when a higher layer delivers a potentially large chunk of data into large low-layer buffers, the higher layer thinks the data is sent out but the data will stay in the buffer for a long time. In fact the higher layer has no way to know when the data actually leaves the device. As indicated in Figure 10, as the location of measurement moves to higher layers, the overall on-device buffer size increases, leading to worse estimation accuracy.

5. IMPACT OF UPLOAD ON MOBILE APPLICATION PERFORMANCE

This section quantifies the impact of upload on some popular applications' performance: file download, web browsing, video streaming, and VoIP. We compare *user-perceived* application performance in two scenarios: without and with concurrent upload. The experiments in this section were conducted on a Samsung Galaxy S3 phone using Carrier 1's LTE network unless otherwise mentioned. We use a single TCP connection to generate upload traffic in controlled experiments.

5.1 Impact of Upload on Bulk Download

When upload and download exist concurrently, upload traffic can affect download traffic in two ways: *in-network* and *on-device*. The former is well-known [47]: upload data shares the same network link with TCP ACK packets of download data, leading to potentially delayed uplink ACK for download. This can cause the server to retransmit download data and reduce the congestion window size, ultimately leading to lower download throughput. On the other hand, the on-device queuing delay triggered by upload can also severely affect download by delaying its ACK packets (shown as t_1 in Figure 6(b)), since when download and upload traffic coexist, uplink TCP ACKs share the same queues (*e.g.,* Qdisc and firmware buffers) with uplink data, as detailed in §4.2.

We carried out experiments of running a one-minute TCP download flow with and without a concurrent TCP upload flow on different devices and carriers with the setup described in §2.1. Table 1 quantifies the impact of upload on download in four aspects, using bulk download in absence of upload as the baseline: *(i)* decrease of the average (AVG) download throughput, *(ii)* increase of the relative standard deviation (RSD)[5] of download throughput, *(iii)* increase of AVG RTT, and *(iv)* increase of RSD of RTT.

[5]Relative standard deviation (RSD) = standard deviation / mean.

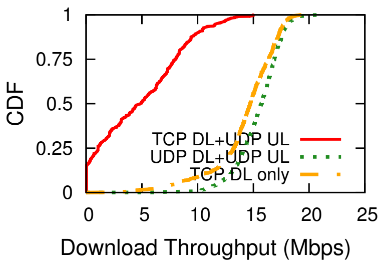


Figure 11: Impact of Uplink traffic on downlink TCP/UDP throughput.

All carriers exhibit performance degradations in various degrees. In particular, large fluctuation of throughput and RTT exists when there is background upload, posing challenges for user-interactive applications. We also compare the in-network and the on-device impact of upload on download traffic, by computing t_1/t_2 in Figure 6(b). The mean and median values of the fractions of t_1 in t_2 are as high as 63% and 74%, indicating *the on-device queuing delay dominates the overall RTT of download traffic.*

Next, we show that when uplink and downlink traffic are both present, the uplink ACK packets being delayed is the dominating cause of degraded download performance. Figure 11 plots the download throughput distributions in three scenarios using Carrier 1's LTE network: *(i)* TCP download only, *(ii)* TCP download with concurrent UDP upload, and *(iii)* UDP download with concurrent UDP upload. Figure 11 indicates that *(i)* and *(iii)* exhibit similar download performance (scenario *(iii)* is even slightly better because it uses UDP for download) while the throughput in Scenario *(ii)* is much lower. Since the key difference between *(ii)* and *(iii)* is whether the uplink ACK stream exists, the results indicate that *the degraded download performance is almost solely associated with TCP's upstream ACKs*, whereas in the underlying radio layer, uplink and downlink use different channels and can be performed independently. Similar results are observed for upload performance (figure not shown).

5.2 Impact on Web Browsing

We next examine the impact of upload traffic on web browsing. We picked ten popular websites from Alexa top sites, and loaded each of them in Google Chrome browser on a Samsung Galaxy S3 phone in two settings: without and with concurrent upload. We repeat the test of each website for 5 times in a row and report the average results. We performed cold-cache loadings for all sites, and measured the page load time (PLT) using QoE Doctor [14, 32].

We found that upload traffic significantly inflates most delay components. For example, the connection setup delay, which usually takes only one round-trip, increases by 64% to 509% due to on-device bufferbloat as the dominating factor. A similar case happens to HTTP requests, which can typically fit into one single TCP packet. HTTP responses that carry downlink data are also affected due to the explanations described in §5.1. The response duration inflates by up to 3464%. Overall, the increase of PLT across the 10 web-

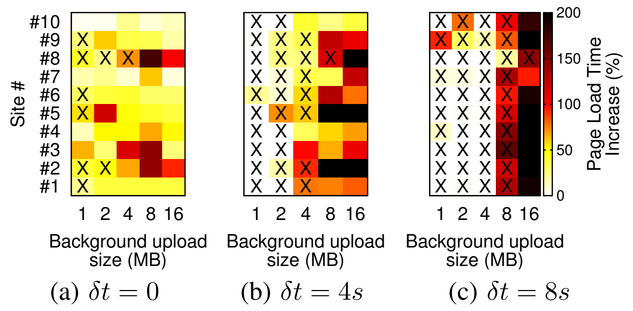


Figure 12: Impact of upload on PLT. The web browsing session begins δt after upload starts. "X" indicates the upload is completed before the web page is fully loaded.

sites ranges from 219% to 607%. The results indicate when concurrent upload is in progress, on-device bufferbloat can significantly affect short-lived flows.

Next, we show that *even a medium-sized upload can cause significant degradation of user experience*. Figure 12 repeats the above experiments but uses a finite size of upload starting at δt seconds before the web browsing session begins. In each subfigure, a heatmap block (x, y) visualizes the PLT inflation caused by an upload of size x for website y. An "X" mark indicates the upload is completed before the page is fully loaded or even started to load so the measured PLT increase is an under-estimation. We observe two trends. First, a larger upload incurs a higher impact on PLT. Second, the PLT impact also becomes higher as δt increases (for blocks without "X"). This is because a larger δt allows more time for the on-device queue to build up, and thus worsens the on-device bufferbloat condition when the web browsing session starts.

5.3 Impact on Video Streaming and VoIP

Video Streaming. We randomly chose 10 popular videos of various lengths (from 40 seconds to 6 minutes) from YouTube and played them over LTE on a Samsung Galaxy S3 phone. The playback software is ExoPlayer [2], which uses the standard DASH streaming algorithm. When the signal strength is above -98dBm, the average playback bitrate across 10 videos is 0.93Mbps without any stall when no concurrent traffic is present. With concurrent TCP upload, the average bitrate is reduced by 57% to only 0.39Mbps, with 15.3 stalls (total stall duration 103s) for each video on average. Even when periodical upload is in progress (upload 5MB data every with 5s idle time between consecutive uploads), half of the videos exhibit playback bitrate degradation by up to 19%.

VoIP. We make Skype voice calls from a Samsung Galaxy S3 phone to a desktop in three settings (3 runs each setting): *(i)* Skype call only, *(ii)* Skype call with concurrent TCP upload, and *(iii)* Skype call with periodical TCP upload of 5MB with 5s idle time between uploads. The experiments were conducted over Carrier 1's LTE network. For each call, we play the same pre-recorded audio (90 seconds) as the baseline and record the audio at the receiver. To quantify the

Bufferbloat Mitigation Solution	Reducing queuing delay			Cross-flow control
	Qdisc	Driver	Firmware	
Change TCP buffer size	(✓)	(✓)	(✓)	
TCP Congestion Control (CC) [19, 12, 11]	(✓)	(✓)	(✓)	
TCP Small Queue (TSQ) [7]	✓			
Traffic prioritization (TP)	✓			✓
Active Queue Management (AQM) [30, 34]	✓			
Byte Queue Limit (BQL) [1]		✓		
QCUT	✓	✓	✓	✓

Table 2: Summary of solutions for reducing queuing delay for upload traffic. "(✓)" means only limited support.

user experience, we use an existing tool [4] to compute the PESQ MOS (Perceptual Evaluation of Speech Quality, Mean Opinion Score) [5] metric. When upload is not present, the average PESQ MOS score is 4.08. With continuous TCP upload, the average PESQ MOS score drops to 1.80. Even for scenario *(iii)*, the average PESQ MOS score is only 1.77.

6. QCUT: SOLUTION FOR ON-DEVICE BUFFERBLOAT

Given the severity of on-device bufferbloat, we propose our solution called QCUT to mitigate it.

6.1 Inadequateness of Existing Solutions

In the literature, numerous solutions have been proposed to mitigate in-network bufferbloat, and some do work with on-device buffers. Table 2 lists representative solutions: changing TCP buffer size, changing TCP congestion control (CC), TCP Small Queue (TSQ), Traffic Prioritization (TP), and Active Queue Management (AQM). However, they all have limitations on reducing on-device queuing delay. Changing TCP buffer size and CC are transport layer solutions that adjust TCP behaviors to reduce the delay. However, they do not work with buffers below the transport layer; also they do not provide cross-flow control as each TCP connection has a separate buffer. TSQ, a newly introduced Linux kernel patch, only limits the Qdisc occupancy on a per-connection basis. TP works across flows and improves user experience by prioritizing delay-sensitive traffic. However, it only partially reduces the queuing delay as will be evaluated in §7.1. The AQM approaches (*e.g.,* CoDel and PIE) also work across flows. But they do not help reduce the buffer occupancy at the firmware buffer. In §7.1, we quantitatively compare all above approaches. We also show that jointly applying them may further incur unexpected conflicts, causing additional performance degradation.

We emphasize that *none of the above solutions can be realized at the firmware buffer*, which is usually proprietary hardware making it difficult to incorporate different queue management algorithms. As new wireless technologies and radio chipsets emerge (*e.g.,* 5G and IoT devices), modification to all firmware to solve the on-device queuing is impractical. Also, as shown in §4.2, the cellular firmware buffer differs from upper-layer buffers in that it plays a role in the cellular control plane (*i.e.,* the BSR affects uplink scheduling and the LTE uplink throughput). Therefore, even ignoring the implementation issues, naïvely applying existing bufferbloat mitigation solutions on the firmware buffer may lead to unexpected results or performance degradation.

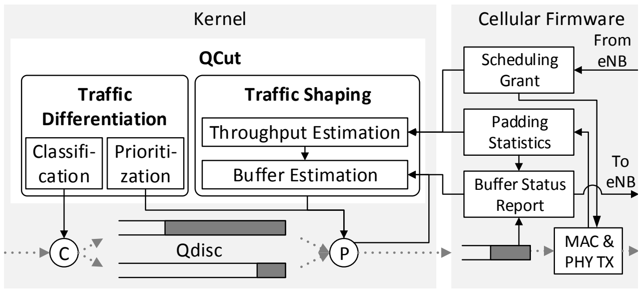


Figure 13: The QCut design.

6.2 QCUT Design

Motivated by the above, we designed and implemented a new approach called QCUT to reduce the on-device queuing delay. Here we focus on optimizing cellular uplink but the general concept of QCUT applies to other networks. As illustrated in Figure 13, QCUT has three prominent features.

- Realized as a general OS service, QCUT is independent of firmware implementation. Therefore it can address the on-device queuing problem on any radio firmware, where no modification is needed. QCUT operates in the kernel space and takes as input only information of buffer occupancy and transmission statistics, which is exposed by most cellular radio firmware from Qualcomm and likely other vendors.
- Since directly limiting the firmware buffer occupancy is difficult, QCUT controls the firmware queuing delay *indirectly in the kernel* by controlling how fast packets from Qdisc flow into the firmware buffer. QCUT estimates the radio firmware buffer occupancy and queuing delay to decide the transmission of packets to the firmware dynamically.
- QCUT is flexible on traffic classification and prioritization. By (indirectly) limiting the amount of data in the firmware, packets are queued in the Linux Qdisc, where QCUT can flexibly prioritize packets based on the application requirements. For example, when background upload and interactive traffic co-exist, the latter can be prioritized and transmitted without Qdisc queuing. By contrast, directly realizing fine-grained traffic prioritization in the firmware is impractical and inflexible.

QCUT aims at reducing the on-device queuing delay. When there is no on-device queuing, QCUT does not incur additional delay to RTT_B or other runtime overhead.

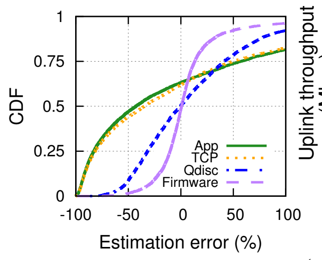


Figure 14: Uplink throughput prediction error at different layers with 20ms prediction interval.

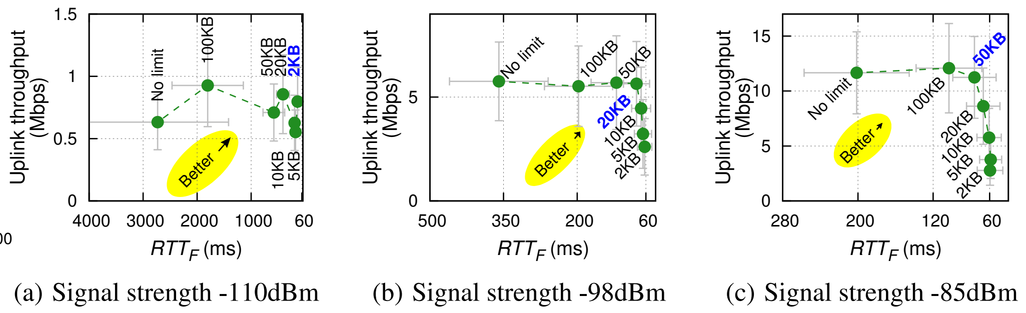


Figure 15: Impact of firmware buffer occupancy threshold of QCUT-B. The best threshold in each plot is in bold blue text.

As shown in Figure 13, QCUT comprises of two components: traffic differentiation and traffic shaping. *Traffic differentiation* classifies packets from applications, and prioritizes certain traffic in the Qdisc (*e.g.,* delay-sensitive traffic) based on applications' requirement. The *traffic shaping* module *(i)* performs accurate throughput prediction, which is then used to *(ii)* estimate the buffer occupancy in the firmware. Based on that, the module *(iii)* controls how fast packets from Qdisc flow into the firmware buffer, in order to limit the firmware buffer occupancy. We describe each component in details below.

Achievable physical layer throughput prediction. Based on recent lower-layer throughput values measured from scheduling grant and padding (§4.4), we perform throughput prediction using Exponentially Weighted Moving Average (EWMA) with $\alpha = 0.25$ (empirically chosen). The prediction interval is 20ms. Note that we need to predict the throughput because the lower-layer firmware information is not provided in real time so the throughput measurement is delayed, as we explain shortly. The "firmware" curve in Figure 14 plots the prediction error distributions under 20ms prediction interval, in our controlled bulk upload experiments with -95dBm RSRP (8Mbps uplink bandwidth). The ground truth is the lower-layer throughput measured with a delay ($\sim$100ms later). The results indicate that compared to other curves in Figure 14 where we perform throughput estimation at higher layers using the same EWMA algorithm, using lower-layer information for throughput prediction is much more accurate.

Buffer occupancy estimation. For a wide range of cellular firmware, their buffer occupancy level can be directly read from the buffer status report (BSR). However, a practical issue we found is that, BSR is not reported in real time to allow accurate buffer occupancy estimation. On both Samsung Galaxy S3 and Nexus 5 devices, although BSR is reported to eNodeB every 5ms, there is on average around 100ms delay before this information is reported to the kernel due to various overheads. During this period, the firmware buffer dynamics may fluctuate considerably.

To overcome this issue, we propose to combine the BSR and the predicted throughput to derive accurate firmware buffer occupancy. The basic idea is the following: since we

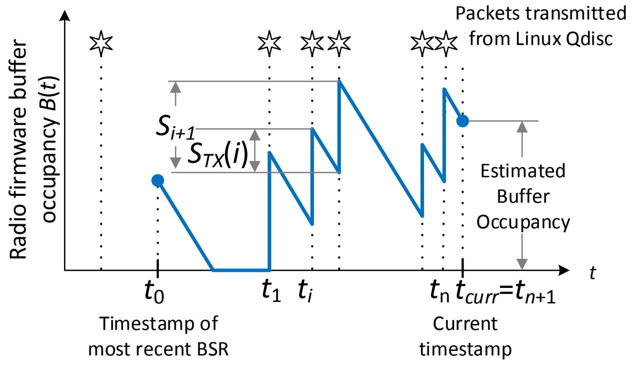

Packets transmitted from Linux Qdisc
Radio firmware buffer occupancy $B(t)$
S_{i+1}
$S_{TX}(i)$
Estimated Buffer Occupancy
t
t_0
t_1
t_i
t_n
$t_{curr}=t_{n+1}$
Timestamp of most recent BSR
Current timestamp

Figure 16: Radio firmware buffer occupancy estimation.

know both the accurate enqueue rate (measured from Qdisc) and dequeue rate (from uplink throughput) of the firmware buffer, we can use them to refine the rough buffer occupancy estimation from delayed BSR. More specifically, let S_0 be the most recently reported BSR generated by the firmware at t_0, which can be obtained from a BSR's timestamp field. Let R_{uplink} be the predicted uplink throughput at t_0. Also we keep track of packets $\{P_i\}$ (i=1,2,...) leaving Qdisc after t_0 by recording their sizes $\{S_i\}$ and timestamps $\{t_i\}$ of leaving Qdisc. Given the above information, the firmware buffer occupancy $B(t_{curr})$ at timestamp t_{curr} can be calculated as follows:

$$B(t_{curr}) = B(t_{n+1}) \quad (1)$$

$$B(t_{i+1}) = B(t_i) + S_{i+1} - S_{TX}(i), i \in [0, n] \quad (2)$$

$$S_{TX}(i) = min(B(t_i), R_{uplink} \times (t_{i+1} - t_i)), i \in [0, n]; \quad (3)$$

where $t_0 < t_1 < .. < t_n \leq t_{n+1} = t_{curr}, S_{n+1} = 0$.

The buffer occupancy is estimated in an iterative manner as shown in Equation (2), where S_{i+1} and $S_{TX}(i)$ are the number of bytes enter and leave the firmware buffer since t_i, respectively. $S_{TX}(i)$ is computed in Equation (3) using the predicted throughput. The process is illustrated in Figure 16.

Qdisc dequeue control. QCUT limits the queuing delay in the radio firmware by throttling the Qdisc in the kernel, *i.e.,* strategically controlling whether a packet should be dequeued from Qdisc into the radio firmware. To realize this, a simple way is to use a fixed threshold of the firmware buffer occupancy, which we refer as QCUT-B (B stands for "bytes"). We evaluated this approach by repeating one-minute TCP uploads five times with different thresholds on a Nexus 5 phone using Carrier 1's LTE network, under different signal strength conditions. As shown in Figure 15, different QCUT-B thresholds incur different tradeoffs between throughput and latency (quantified by RTT_F). However, it is difficult to find a threshold that works for all network conditions. The best threshold that achieves low latency without sacrificing the throughput depends on the signal strength: 2KB for -110dBm, 20KB for -98dBm, and 50KB for -85dBm. In particular, a small threshold (*e.g.,* 2KB) works well when the signal strength is low. However, at high signal strength, it causes bandwidth under-utilization. As described in §4.2, this is attributed to the very nature of cellular uplink scheduling: the firmware buffer occupancy reported in BSR is used for determining uplink bandwidth allocation; the base station thus regards a small buffer occupancy as an indicator that the client does not have much data to transmit, thus allocating small uplink bandwidth for the mobile client.

To overcome the above limitation, we propose another scheme called QCUT-D (D stands for "delay"). It instead uses the firmware queuing delay (T_F) as a threshold. The queuing delay is computed from the estimated throughput and the buffer occupancy. If the delay is above the threshold, QCUT-D does not allow a packet to be dequeued to the firmware from Qdisc. Thus, QCUT-D is adaptive to diverse network conditions by dynamically adjusting the firmware buffer occupancy. We empirically found that using 20ms as the delay threshold on LTE networks works reasonably well in diverse network conditions: it leads to low firmware buffer queuing while incurring very small impact on the uplink throughput, as to be evaluated in §7.2. This threshold can also be empirically chosen for other types of networks.

Traffic differentiation. To meet the performance requirement of different applications, QCUT uses the priority queuing in Linux Qdisc for traffic prioritization. For example, the background upload and interactive traffic such as web browsing are put into different queues in Qdisc. As a result, interactive traffic does not experience high queuing delay in Qdisc caused by bulk upload. Also, thanks to the aforementioned traffic shaping module in QCUT, the delay-sensitive traffic also undergoes very low queuing delay in the firmware, thus leading to an overall small on-device queuing delay and thus good user experience. QCUT uses existing traffic classification mechanism on Linux to allow applications and users to flexibly configure priorities for different traffic through the standard `tc` interface.

6.3 QCUT Implementation

We implemented QCUT on Android Linux kernel. Our testing devices consist of Samsung Galaxy S3 and Nexus 5 running Android 4.4.4 and 6.0.1 with Qualcomm radio chipset. We expect QCUT to also work with other phones and tablets with cellular firmware from the same vendor. Note that QCUT does not require any special equipment such as QXDM [6].

Traffic shaping and differentiation are implemented as a Linux packet scheduler module in 600 LoC. QCUT keeps track of transmitted packets from Qdisc since the most recent BSR. The traffic shaping module is implemented in the function call `enqueue()` of the Qdisc operation data structure `Qdisc_ops`. In `enqueue()`, the queuing delay in firmware is estimated based on the information from the radio firmware. More specifically, we use the `/dev/diag` interface on the Android phones with Qualcomm radio chipset to extract the uplink scheduling grant, padding statistics, and BSR from the logs of LTE uplink transport blocks. The online parsing of the logs is implemented in a C++ program in the user space. Each log record has a timestamp of the firmware. The timestamps between kernel and the firmware need to be synchronized. The user-space program sends time request periodically and uses the response, which contains the firmware timestamp, to perform the synchronization.

7. EVALUATION

We comprehensively assess how a wide range of solutions help mitigate the on-device bufferbloat problem, focusing on existing solutions (§7.1) and then QCUT (§7.2). For all the following experiments, we conducted on a Samsung Galaxy S3 on Carrier 1's LTE network unless otherwise mentioned. We expect the experimental findings to be general as none of the solutions depends on a specific carrier or vendor.

7.1 Existing Solutions

We consider existing bufferbloat-mitigation solutions discussed in §6.1. We demonstrate in this section that they can reduce excessive on-device queuing to various degrees. However, they suffer from various limitations, and are all incapable of reducing the firmware buffer occupancy. We conduct bulk upload experiments at two locations with different signal strengths measured by RSRP (Reference Signal Received Power): good signal (RSRP of -69 to -75 dBm) and fair signal (RSRP of -89 to -95), using a Samsung Galaxy S3 on Carrier 1's LTE network.

Changing TCP buffer sizes. The TCP send buffer (`tcp_wmem`) on device imposes a limit on the TCP congestion window (cwnd). As shown in Figure 17, under good signal, shrinking the send buffer effectively reduces RTT_Q that is dominated by device-side queuing at Qdisc and firmware. However, the penalty is severely degraded upload throughput, in particular when `tcp_wmem` is smaller than the bandwidth-delay product (BDP). Since BDP constantly fluctuates in cellular networks [43], a fixed configuration of TCP buffer size does not fit all network conditions.

Changing TCP small queue (TSQ) size. As a newly introduced Linux kernel patch, TSQ [7] limits per-connection data in Qdisc using a fixed threshold. By reducing the threshold, we observe smaller Qdisc queuing delay (T_Q = $RTT_Q - RTT_F$ in Figure 8) under both network conditions, as shown in Figure 18. Yet Linux's default TSQ threshold is too large to eliminate the Qdisc queuing. However, Fig-

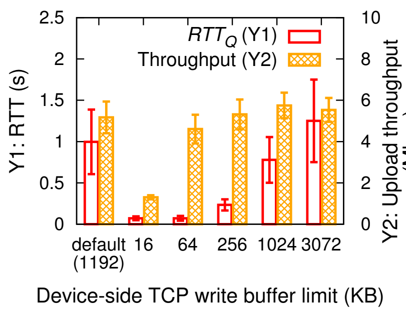


Figure 17: Impact of TCP send buffer sizes on upload performance.

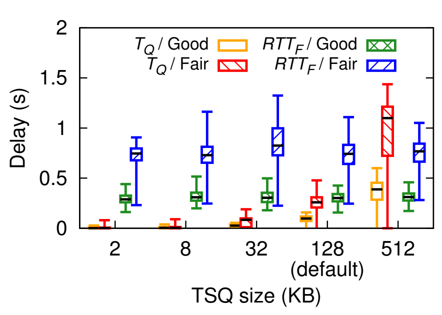


Figure 18: Impact of different TCP small queue (TSQ) sizes on upload.

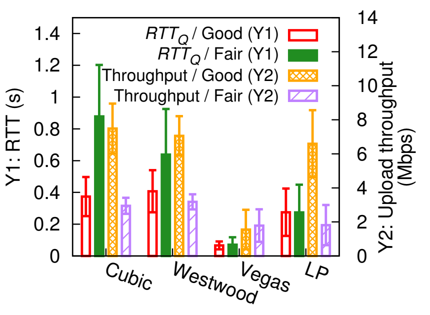


Figure 19: Impact of different TCP CC on upload (with TSQ=128KB).

ure 18 also indicates that TSQ has negligible impact on the firmware queuing delay ($T_F = RTT_F - RTT_B$), because TSQ only controls the bytes in Qdisc. Further, TSQ limits its Qdisc occupancy on a *per-connection* basis so the Qdisc occupancy can still be high when concurrent flows exist.

Changing TCP congestion control. Congestion control (CC) affects the aggressiveness of TCP. We consider two representative CC categories: loss-based CC (TCP CUBIC[19] and Westwood[29]) and delay-based CC (TCP Vegas[12] and LP[24]). Generally speaking, lost-based CC, which uses packet loss as congestion indicator, is more aggressive than delay-based CC that treats increased delay as a signal of congestion. We found even with TSQ enabled, loss-based CC incurs severe on-device queuing, measured by RTT_Q, as shown in Figure 19. For delay-based CC, regardless of TSQ setting, on-device queuing is almost always negligible. However, such low on-device queuing delays are achieved by sacrificing up to 80% of the throughput.

Active Queue Management is a major in-network solution to reduce queuing delay and network congestion by strategically dropping packets in a queue. We considered two well-known and recently proposed AQM algorithms, CoDel [30] and PIE [34]. Both approaches use a target threshold to control the queuing delay. Under both signal strengths, CoDel effectively keeps the Qdisc queuing delay below the target threshold. However, Since CoDel does not apply to the firmware buffer, it only slightly reduces RTT_F by 10% to 20%, as indicated in Figure 20. This is the result of TCP cwnd reduction triggered by packet losses injected by CoDel. The performance of PIE is even worse than CoDel.

Jointly applying multiple strategies. In many case, several mitigation strategies can be jointly applied to better balance various tradeoffs. However, we find that jointly using several approaches may also incur unexpected conflicts, causing performance degradation. For example, when CoDel (with target threshold 5ms) and TSQ (with queue size 4KB) are jointly applied to a single upload flow, RTT_F actually increases by 37% compared to using CoDel alone (figure not shown). This is explained as follows. A small Qdisc achieved by TSQ can reduce the effectiveness of CoDel, since the small on-device queuing delay allows CoDel to drop very few packets compared to a large queue does. This causes TCP cwnd to increase faster, leading to more noticeable in-network queuing delay.

Traffic prioritization. All above solutions focus on reducing on-device queuing for bulk upload. When concurrent upload and download exist, an alternative approach is to prioritize uplink ACK packets over upload data traffic to mitigate the impact of upload on download (§5.1). Our experiments indicate that when uplink ACKs are prioritized, their Qdisc queuing delay is reduced significantly from 1363ms to 86ms at -95dBm. However, prioritization can only be realized at Qdisc, causing the uplink ACK stream still to interfere with uplink data at the firmware buffer. As a result, compared to the case where no concurrent upload exists, applying prioritization still increases RTT_F by 112ms. We will demonstrate in §7.2 that by combining Qdisc prioritization with firmware queuing delay reduction, QCUT can effectively mitigate on-device bufferbloat.

7.2 Evaluation of QCut

We conduct a thorough evaluation of QCUT to demonstrate that it outperforms existing solutions. First, we show QCUT can significantly reduce RTT_F that mainly consists of the firmware queuing delay. We then conduct a crowd-sourced user study to assess the effectiveness of QCUT under real workload (web browsing and video streaming) when bulk upload is present.

Reducing excessive firmware queuing. Using the workload of a single TCP upload, we compare the performance of five schemes: TCP CUBIC, TCP Vegas, TSQ, CoDel, and QCUT. Each experiment thus consists of five back-to-back TCP uploads (one minute each) using Carrier 1's LTE network. We repeat the experiment for 10 times at two locations with stable signal strength of -95dBm and -110dBm, respectively. We calculate the throughput every 500ms and measure RTT_F using `tcpdump` traces. For each scheme, we report the average result at each location.

Since the five schemes achieve different tradeoffs between throughput and latency, we visualize the results on a two-dimensional plane in Figure 21. The X and Y axes correspond to RTT_F and measured throughput, respectively. A good solution should appear in the upper-right corner of the plane. The results indicate that except for QCUT and TCP Vegas, none of the five solutions is capable of reduc-

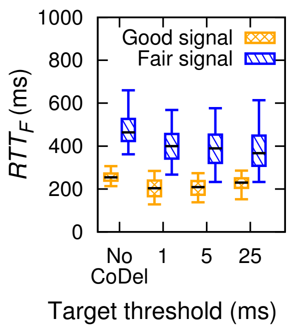
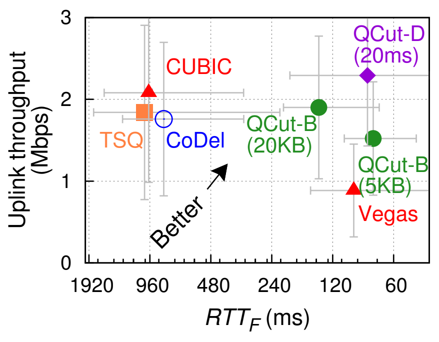
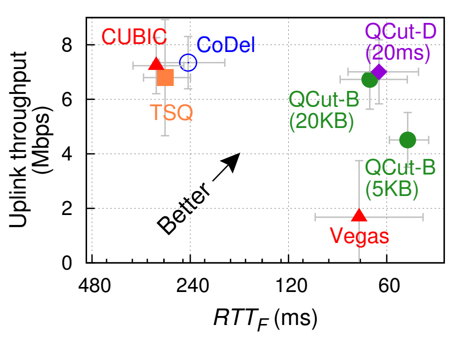


Figure 20: Effectiveness of CoDel on reducing latency.

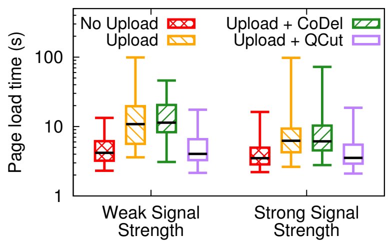


(a) Signal Strength -110dBm

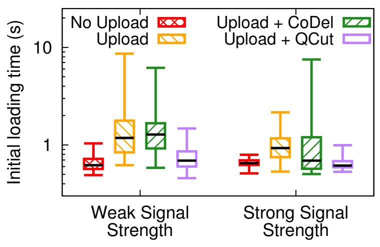


(b) Signal Strength -95dBm

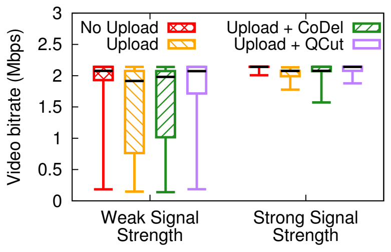
Figure 21: Compare TCP upload performance of different schemes.



(a) Web browsing



(b) Video streaming: initial loading time



(c) Video streaming: bitrate

Figure 22: Improvement of application performance brought by QCUT.

ing RTT_F because they do not work at the firmware layer. For TCP Vegas, in Figure 21(b), it achieves low latency at the cost of very low throughput, with the reason explained in §7.1. On the other hand, QCUT effectively reduces the firmware queuing with little or small sacrifice of the throughput. Recall in §6.2 that we devised two QCUT schemes: QCUT-B and QCUT-D, which use the firmware buffer *occupancy* and *delay* as the threshold to limit the firmware buffer occupancy. We found QCUT-D works reasonably well at both locations since it is adaptive to different throughput, while it is a bit difficult to pick a fixed threshold for QCUT-B for different throughput.

Improving application performance. To assess how QCUT improves real applications' performance, we deployed QCUT on five Samsung Galaxy S3 phones used by real users. The phones run crowd-sourced measurements supported by Mobilyzer[33] for a week under diverse network conditions. This user study has been approved by IRB.

We consider two workloads: (1) load five popular webpages, and (2) stream a 2-min YouTube video. For each workload, we run back-to-back measurements under four different settings: *(i)* no background upload, *(ii)* concurrent upload without bufferbloat mitigation, *(iii)* concurrent upload with CoDel on Qdisc, and *(iv)* concurrent upload with QCUT-D. For web browsing, we collect the page load time (PLT) of each webpage; for video streaming, we record initial loading time, playback bitrate and rebuffering events. Note we only triggered these measurements when the phone is idle, so the experiment is not interfered with other user traffic. To mitigate the impact of the varying signal strength within the same experiment that consists of four back-to-back measurements, we discard the entire experiment if the LTE RSRP changes by more than 4dBm. Overall we conducted 1266 and 549 successful experiments for web browsing and video streaming, respectively.

The results are shown in Figure 22. In each plot, we show two groups of results corresponding to weak signal strength (LTE RSRP<-99dBm) and strong signal strength (LTE RSRP≥-99dBm), respectively. As shown in Figure 22(a), due to concurrent upload, the median PLT across 5 sites increases by 78% and 159% for strong and weak signal strength, respectively, leading to significantly degraded user QoE. Applying CoDel does not help mitigate the additional ACK delay (of webpage download) incurred by the bulk upload, in particular when the signal strength is weak: the median PLT increases are still as large as 75% and 171% for strong and weak signal strength, respectively, compared to the no-upload cases. QCUT-D, on the other hand, effectively reduces the PLT to the baseline (*i.e.,* no-upload). For video streaming, we consider two QoE metrics: initial buffering time and playback bitrate, whose results are shown in Figure 22(b) and 22(c), respectively. Again, QCUT significantly outperforms CoDel on improving the video streaming QoE when concurrent bulk upload is present.

As described in §6.2, the effectiveness of QCUT is attributed to two reasons. First, it reduces the firmware buffer occupancy (T_F). But doing that alone is not sufficient because delay sensitive traffic can still be interfered by upload traffic at Qdisc. QCUT addresses this by performing prioritization at Qdisc, resulting in reduced T_Q for delay sensitive traffic.

8. RELATED WORK

Measuring cellular performance. Several prior efforts focus on characterizing cellular performance at various aspects. Studies [15, 39, 37, 31] collect data from deployed user trials to understand smartphone performance at different layers. To name a few, Sommers *et al.* leveraged `speedtest` data to compare cellular versus Wi-Fi performance for metro area mobile connections [40]. Huang *et al.* examined LTE bandwidth utilization and its interaction with TCP [20]. Shafiq *et al.* conducted a study of cellular network performance during crowded events [38]. Liu *et al.* measured performance of several TCP variants on 3G EvDO networks [28]. Rosen *et al.* studied the impact of RRC state timers on network and application performance [35, 36]. Jia *et al.* performed a systematic characterization and problem diagnosis of commercially deployed VoLTE (Voice over LTE) [21]. None of the above studies deeply examined cellular upload traffic that is becoming increasingly popular.

Improving transport protocols. Over the past 30 years, researchers have produced a large body work on improving TCP. We already mentioned many TCP congestion control algorithms in §6.1, such as [19, 29, 16, 13, 25, 23, 12, 24, 11, 17, 27]. Some recent proposals such as TCP-RRE [26], Sprout [43], Verus [46] and TCP ex Machina [42] leverage throughput forecasts or machine learning to find the optimal data transmission strategy. All these approaches face the problem of balancing between throughput and latency, which is a key factor to be considered when selecting the desired CC given the application requirement. Other solutions like RSFC [44] and DRWA [22] uses receive buffer to limit the queuing impact. However, transport-layer solutions do not provide cross-flow control, which may not fully eliminate interference between flows. Compared to these transport-layer solutions, QCUT uses accurate throughput estimation based on the information from the firmware and explicitly reduces on-device queuing in cellular networks.

Understanding excessive queuing delay ("bufferbloat"). The bufferbloat problem is known in both wired and wireless networks. Gettys *et al.* presented an anecdotal study [18] on large queuing delay of interactive traffic. The study focuses on the scenario of concurrent bulk data transfers in cable and DSL networks. Using real network traces, Allman argued that although bufferbloat can happen, the problem happens more in residential than non-residential networks and the magnitude of the problem is modest [10]. However, the issue is indeed severe in cellular networks that usually employ deep buffers, as shown in a study conducted by Jiang *et al.*, who explored the bufferbloat problem of downlink traffic in 3G and LTE networks [22]. Recent work by Xu *et al.* indicates that some newer smartphones seem to buffer packets in the kernel when UDP packets are transmitted continuously [45]. However, they did not study TCP or consider how application is affected. In contrast, we carry out comprehensive measurements to quantitatively understand *(i)* on-device bufferbloat caused by TCP *upload* traffic and its impact on applications, *(ii)* the interaction between upload and other traffic patterns, *(iii)* the interplay between TCP and lower layer queues, and *(iv)* the effectiveness of a wide range of mitigation strategies at different layers.

Mitigating bufferbloat. Besides those evaluated in §7, there exist other proposals for reducing excessive queuing delay. Dynamic Receive Window Adjustment (DRWA) [22] is a receiver-side solution to reduce the queuing delay by adjusting the TCP receive window. Originally it is deployed on mobile devices to reduce the latency for downlink traffic. For the uplink case, DRWA needs to be deployed at server side that serves both cellular and non-cellular clients, thus posing deployment challenges. Byte Queue Limits (BQL) [1] is another proposal that puts a cap on the amount of data waiting in the device driver queue. It does not apply to Qdisc that contributes the majority of the on-device latency. Moreover, BQL needs driver support.

9. CONCLUDING REMARKS

We carried out to our knowledge the first comprehensive investigation of cellular upload traffic and its interaction with concurrent traffic. Our extensive measurement using 33-month crowd-sourced data indicates the contribution of upload is large, and the upload speed is high enough to enable applications to upload user-generated traffic. We then comprehensively investigated the on-device bufferbloat problem that incurs severe performance impact on applications. We identified a major source of on-device bufferbloat to be the large firmware buffer, on which existing bufferbloat mitigation solutions are ineffective. We then propose a general and lightweight solution called QCUT, which controls the firmware buffer occupancy from the OS kernel. We demonstrate the effectiveness of QCUT through in-lab experiments and real deployment.

Acknowledgements

We would like to thank our shepherd, Mark Allman, and the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Shichang Xu for his assistance in obtaining information from radio firmware. This research was supported in part by the National Science Foundation under grants CNS-1059372, CNS-1345226, CNS-1566331 and CNS-1629894.

10. REFERENCES

[1] Byte Queue Limit. `https://lwn.net/Articles/454390/`.
[2] ExoPlayer. `http://developer.android.com/guide/topics/media/exoplayer.html`.
[3] Introduction to LTE Advanced. `http://www.androidauthority.com/lte-advanced-176714/`.

[4] OPTICOM, PESQ - perceptual evaluation of speech quality. `http://www.opticom.de/technology/pesq.php`.

[5] P.862: Perceptual evaluation of speech quality (PESQ). `https://www.itu.int/rec/T-REC-P.862-200102-I/en`.

[6] Qualcomm eXtensible Diagnostic Monitor. `https://goo.gl/LODgRY`.

[7] TCP Small Queues. `https://lwn.net/Articles/507065`.

[8] `tcp_probe`. `http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe/`.

[9] 3GPP TS 36.321: Medium Access Control (MAC) protocol specification (V10.3.0), 2011.

[10] M. Allman. Comments on bufferbloat. *ACM SIGCOMM CCR*, 2012.

[11] A. Baiocchi, A. P. Castellani, and F. Vacirca. Yeah-tcp: yet another highspeed tcp. In *PFLDnet*, 2007.

[12] L. S. Brakmo and L. L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, 1995.

[13] C. Caini and R. Firrincieli. Tcp hybla: a tcp enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, 2004.

[14] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, , and K. Lau. QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis. In *ACM IMC*, 2014.

[15] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, and R. G. D. Estrin. Diversity in Smartphone Usage. In *ACM Mobisys*, 2010.

[16] S. Floyd. Highspeed tcp for large congestion windows. RFC 3649, 2003.

[17] C. P. Fu and S. C. Liew. Tcp veno: Tcp enhancement for transmission over wireless access networks. *Selected Areas in Communications, IEEE Journal on*, 21(2):216–228, 2003.

[18] J. Gettys. Bufferbloat: Dark buffers in the internet. *IEEE Internet Computing*, 15(3):96, 2011.

[19] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[20] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of LTE: effect of network protocol and application behavior on performance. In *ACM SIGCOMM*, 2013.

[21] Y. J. Jia, Q. A. Chen, Z. M. Mao, J. Hui, K. Sontineni, A. Yoon, S. Kwong, and K. Lau. Performance Characterization and Call Reliability Problem Diagnosis for Voice over LTE. In *ACM MobiCom*, 2015.

[22] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling Bufferbloat in 3G/4G Networks. In *IMC*, 2012.

[23] T. Kelly. Scalable tcp: Improving performance in highspeed wide area networks. *ACM SIGCOMM CCR*, 2003.

[24] A. Kuzmanovic and E. W. Knightly. Tcp-lp: A distributed algorithm for low priority data transfer. In *IEEE INFOCOM*, 2003.

[25] D. Leith and R. Shorten. H-tcp: Tcp for high-speed and long-distance networks. In *PFLDnet*, 2004.

[26] W. K. Leong, Y. Xu, B. Leong, and Z. Wang. Mitigating egregious ack delays in cellular data networks by eliminating tcp ack clocking. In *IEEE ICNP*, 2013.

[27] S. Liu, T. Başar, and R. Srikant. Tcp-illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 65(6):417–440, 2008.

[28] X. Liu, A. Sridharan, S. Machiraju, M. Seshadri, and H. Zang. Experiences in a 3G Network: Interplay between the Wireless Channel and Applications. In *ACM MobiCom*, 2008.

[29] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *ACM MobiCom*, 2001.

[30] K. Nichols and V. Jacobson. Controlling queue delay. *ACM Queue*, 10(5), 2012.

[31] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen. An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design. In *ACM MobiCom*, 2016.

[32] A. Nikravesh, D. K. Hong, Q. A. Chen, H. V. Madhyastha, and Z. M. Mao. QoE Inference Without Application Control. In *ACM SIGCOMM Internet-QoE Workshop*, 2016.

[33] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An open platform for controllable mobile network measurements. In *ACM Mobisys*, 2015.

[34] R. Pan, P. Natarajan, C. Piglione, M. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. Pie: A lightweight control scheme to address the bufferbloat problem. In *IEEE HPSR*, 2013.

[35] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks. In *ACM MobiCom*, 2014.

[36] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Understanding RRC State Dynamics through Client Measurements with Mobilyzer. In *ACM MobiCom S3 Workshop*, 2014.

[37] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen. Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild. In *ACM IMC*, 2015.

[38] Z. Shafiq, L. Ji, A. Liu, J. Pang, S. Venkataraman, , and J. Wang. A First Look at Cellular Network Performance during Crowded Events. In *ACM SIGMETRICS*, 2013.

[39] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: Measuring Wireless Networks and Smartphone Users in the Field. In *HotMetrics*, 2010.

[40] J. Sommers and P. Barford. Cell vs. WiFi: On the Performance of Metro Area Mobile Connections. In *IMC*, 2012.

[41] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband Internet Performance: A View From the Gateway . In *ACM SIGCOMM*, 2011.

[42] K. Winstein and H. Balakrishnan. TCP ex machina: computer-generated congestion control. In *ACM SIGCOMM*, 2013.

[43] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *USENIX NSDI*, 2013.

[44] Y. Xu, W. K. Leong, B. Leong, and A. Razeen. Dynamic regulation of mobile 3g/hspa uplink buffer with receiver-side flow control. In *IEEE ICNP*, 2012.

[45] Y. Xu, Z. Wang, W. K. Leong, and B. Leong. An end-to-end measurement study of modern cellular data networks. In *PAM*, 2014.

[46] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive congestion control for unpredictable cellular networks. In *ACM SIGCOMM*, 2015.

[47] L. Zhang, S. Shenker, and D. D. Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. *ACM SIGCOMM CCR*, 1991.