Trimming the Smartphone Network Stack

Yanzi Zhu, Yibo Zhu[§], Ana Nika, Ben Y. Zhao, Haitao Zheng
University of California, Santa Barbara [§]Microsoft Research
{yanzi, anika, ravenben, htzheng}@cs.ucsb.edu, yibzh@microsoft.com

ABSTRACT

Network transmissions are the cornerstone of most mobile apps today, and a main contributor to energy consumption. We use a componentized energy model to quantify energy use by device, and observe significant energy consumption by the CPU in network operations. We assert that optimizing network operations in the CPU can produce significant energy savings, and explore the impact of two potential approaches: one-copy data moves and offloading the network stack to the basestation.

1. INTRODUCTION

Mobile networking consumes a significant portion of smartphone energy [9, 8]. Much of the today's research on energy efficient mobile networks focuses on system-level approaches, *e.g.,* saving power by shutting down entire components or putting the device to sleep. This is partly due to the fact that, until recently ([9, 8, 27]), researchers had limited visibility into the energy consumed by individual system components. In contrast, quantifying energy consumed by each component can enable the development of component-specific energy-saving techniques.

In this paper, we take a different approach to energy-efficient mobile networking, by first studying the energy consumed by the CPU during network operations using a component-based energy model. We consider the energy model proposed and validated by [27], which we also validated using power meter measurements. Using detailed CPU usage logs and packet traces, this model is able to identify the amount of energy consumed by the CPU during interrupt handling and network stack processing, as a portion of overall energy consumption. We perform energy measurements using this model for several instrumented Android devices running both constant bit rate (CBR) and best effort network applications. Surprisingly, we find that the CPU is responsible for a significant portion of the overall system energy consumption (CPU, NIC, other processes). This is especially true for WiFi at high data rates, where the CPU dominates overall energy by consuming more than 60% of total energy. Even for energy-intensive LTE networks, the CPU still accounts for up to 20% of overall energy usage.

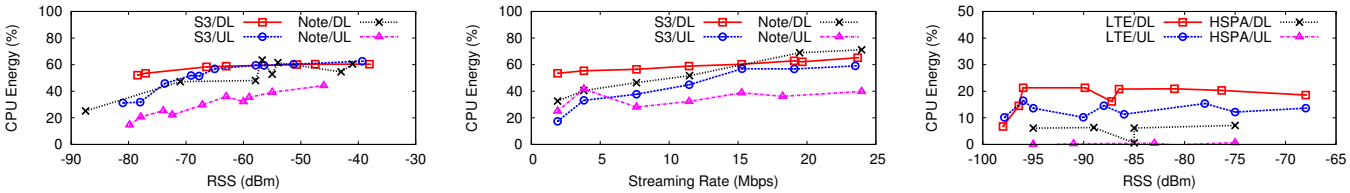
Reducing CPU Energy Usage. We believe these observations warrant a fresh look at the role of CPU usage on today's network-driven mobile devices. Specifically, we examine two different approaches to reduce the significant energy consumption in today's mobile CPUs. *First*, we study the current network stack implementation in Android, and look for ways to reduce CPU load by reducing memory copies. While this is a known technique for improving network performance in wired networks, we believe it will have significant impact on CPU energy consumption in mobile devices.

Second, we consider a much more drastic approach: minimizing CPU load by offloading network stack processing to the wireless basestation. Stack processing includes data copying, packetization, and managing the TCP/IP stack. The intuition for offloading is simple: network stack processing is often complicated and computationally costly, and the resulting end-to-end traffic management (*e.g.,* TCP) performs poorly over wireless links, especially at high data rates.

A wireless device offloads its network stack by communicating directly with its basestation using basic traffic segments. Assuming the basestation is cooperative and has sufficient resources (buffers, processing power), the basestation effectively acts like an end-to-end proxy that manages TCP endpoints for each of its mobile device clients. Not only does the basestation reduce computation at the mobile device, but it eliminates the role of wireless channel loss from TCP congestion management [6], and addresses the problem of ACK contention on the wireless channel [15].

We carefully evaluate both approaches for reducing CPU energy cost. We implement and test the impact of one-copy communication on CPU energy consumption in Android. We then implement TCP offloading for several Android applications, and evaluate its impact on CPU energy consump-

HotNets-XV, November 09-10, 2016, Atlanta, GA, USA
 ISBN 978-1-4503-4661-0/16/11…$15.00
DOI: http://dx.doi.org/10.1145/3005745.3005759



(a) Best efforts (WiFi) (b) Constant bitrate (WiFi) (c) Best efforts (LTE (S3) and HSPA (Note))

Figure 1: The ratio of the CPU-networking energy consumption against the total energy consumption.

tion on WiFi networks for a range of signal strength and transmission rates. Our results are promising: for today's WiFi connections (<24Mbps), one-copy and TCP offload already achieve considerable CPU energy savings, 10% and 40%, respectively. As the wireless capacity continues to increase, (*e.g.,* 100–200Mbps), the estimated gain becomes substantial, 40% for one-copy and 60% for TCP offload.

As a first step, our results show that additional work is warranted to further understand the pros and cons of TCP wireless offloading as an energy management tool. We discuss unanswered questions and future directions in §7.

2. CPU ENERGY CONSUMPTION

With the recent availability of componentized power models, we can now quantify the energy consumed by individual components of a mobile device [29]. Here, we study the role of the CPU in energy consumption due to network operations, by measuring energy consumed by the CPU during WiFi or LTE transmissions. We show that, surprisingly, the CPU often consumes even more energy than the wireless NIC, especially for WiFi transmissions. This suggests that significant energy savings can come through optimizing CPU operations due to network transmissions.

2.1 Measurement Setup

Our analysis uses the componentized energy models [27] to measure the energy consumed by the CPU and wireless NICs. The models estimate the detailed energy consumption of each component from CPU usage logs and packet traces captured by *tcpdump*. Prior work shows that the models can estimate energy consumption of individual components, and incurs less than 8% error compared with physical power meter readings [27]. Using a Monsoon power meter, we also validate the energy models ourselves (see Appendix).

Our measurements use two popular Android smartphones, *Samsung Galaxy S3* with Android 5.1.1 and *Samsung Galaxy Note* with Android 4.3. We obtained detailed power models for both phones from the authors of [27]. Our measurements target Android because it is flexible to instrument and profile. Since implementations of the network stack on different smartphone platforms generally follow the same RFC standards, we expect that our observations will apply to other platforms. We leave this to future work.

CPU and Packet Logging. We build a logging tool that records the per-core CPU frequency and utilization reported by the Android OS (once every $500ms$), and run *tcpdump* to collect packet traces. The logging overhead is low — running it with *tcpdump* simultaneously only increases the CPU utilization by at most 5%.

Baseline Energy Consumption. We verify that without running any networking application, the CPU utilization is at most 7% (for all background OS processes). We use the power meter to measure the total power consumption during this state. The S3 phones consume *150.2mW* and the Note phones consume *203.9mW*. In our analysis, we subtract this value (referred to as E_{bk}) from the CPU energy results to reflect the CPU energy consumption *purely* due to networking.

Measurement Scenarios. We consider two common network-intensive applications: *constant bit rate (CBR) video streaming* and *best effort file transfer*, *e.g.,* photo/video sharing. We implement both types of traffic workloads in our bareboned network apps, which only transfer data, and experiment with video/file sizes between 1MB and 100MB. For CBR streaming, we use streaming rates between 1Mbps and 24Mbps supported by today's streaming services [1].

We include experiments on both WiFi (802.11n on 2.4GHz) and cellular networks (AT&T LTE and HSPA). We test both downlink and uplink transmissions, for a range of (average) signal strengths (-90dBm to -30dBm for WiFi and -100dBm to -65dBm for cellular). To reduce interference and contention, we configure our WiFi AP away from existing active channels; for cellular tests, we perform measurements after midnight to reduce the impact of traffic dynamics at the basestation. For statistical significance, we repeat each of our experiments at least 10 times. Since standard deviation across measurements is uniformly low (less than 2.5%), we only show average results in our discussion.

2.2 Results

We first compare the energy consumption of the CPU and the wireless NIC due to networking. Figure 1 plots, for both WiFi and cellular, energy consumed by the CPU (specifically for networking) as a portion of total energy cost of the NIC, CPU (for networking), and OS background processes. Specifically, we study the normalized CPU energy consumption, $\frac{E_{CPUnet}}{E_{CPUnet}+E_{NIC}+E_{bk}}$. Because results are consistent across various data sizes, for brevity we only show the results for sending 100MB data.

For WiFi transmissions, the CPU is a major energy consumer, contributing to 20–60% of the total energy. The portion for uplink is lower because the NIC consumes more power in transmission mode. For best effort data transfer,

CBR Rate	Logging	OS Proc.	WiFi Driver	Network Stack
4Mbps	1.7%	3.3%	10.2%	31.5%
24Mbps	2.3%	4.6%	15.6%	43.2%

Table 1: CPU utilization during WiFi CBR streaming

CPU energy cost dominates more at higher RSS values: as the link rate increases, physical transmissions become more energy-efficient (more bits per Joule) while the CPU processes more data. The same applies to CBR streaming at higher streaming rates, *i.e.* CPU must process more data at higher streaming rates. Finally, even for power-hungry LTE links, the CPU consumes up to 20% of the total energy.

CPU Usage Breakdown. Next we quantify how different tasks contribute to CPU usage. In our scenarios, these tasks include processing the network stack, running our logging app and *tcpdump*, processing the NIC driver, and OS background processes. We measure their contribution by recording the per-process CPU utilization using our logging tool. Our results are consistent — network stack processing is the dominant CPU consumer. Table 1 shows the CPU utilization of different components for WiFi CBR video streaming. Network stack processing is responsible for more than 50% of the CPU activities. We also confirm via experiments that CPU energy consumption is determined jointly by CPU utilization and CPU frequency[1]. In fact, the energy consumption scales linearly with both CPU frequency and utilization.

Together, these results indicate that the CPU usage and energy consumption are dominated by network stack processing. Simplifying the CPU's role in network stack processing will likely provide significant energy savings.

3. REDUCING CPU LOAD

Motivated by the above findings, we revisit the Android network stack processing to identify ways to reduce CPU usage. We show that without modifying the wireless NIC, there are two potential directions, *i.e. reducing local data copy* and *TCP offload*. While both methods have been proposed to improve throughput for high speed networks such as Gigabit Ethernet, we seek to evaluate their effectiveness on reducing energy costs for smartphones.

Android Network Stack Processing. The Android network stack processing consists of three sequential tasks: *copying data* between the network (socket) buffer and other buffers (file, application); *processing Layer 2 to Layer 4 protocols*; and *packetization*. As an example, Figure 2 illustrates the stack processing during data upload, where an application seeks to upload data stored on the smartphone to a remote server via WiFi, *e.g.,* uploading an image or video to the server. In this case, after DMA buffers the data to the kernel, the kernel copies data to the userspace (application buffer) and then back to the socket buffer. Data then goes through the network stack processing, and finally is packetized for the NIC to send out. Download follows a similar process.

We identify two sources of inefficiency in this design. *First*, before entering the NIC, the target data is being copied twice (between the kernel and userspace). In this process, the application acts as an inefficient intermediary that fetches the data from the disk storage to the socket. This is unnecessary and inefficient. *Second*, current TCP stack on smartphones is not designed with energy efficiency in mind. For example, common TCP implementations ACK every one or two packets. In addition to introducing extra processing overhead, these ACK packets also contend with data packets, further lowering wireless bandwidth utilization (as the NICs are not full-duplex). As a result, the active time of the CPU is longer, leading to higher energy consumption.

[1]Modern mobile OS actively adjusts CPU frequency based on the total workload of all running processes.

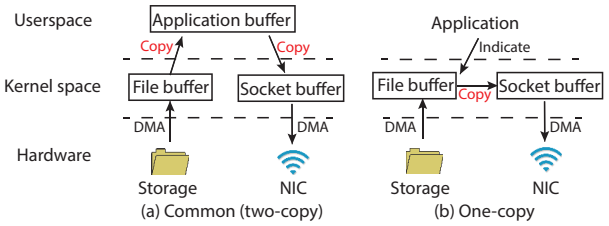


Figure 2: One-copy improve CPU efficiency.

With these in mind, we explore two directions that reduce CPU load by minimizing network stack processing.

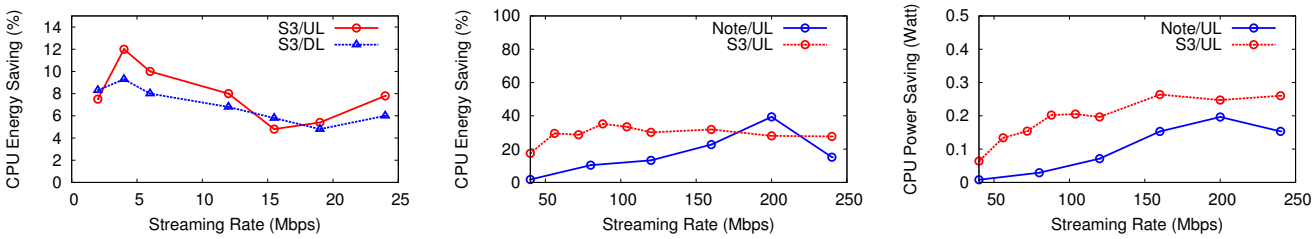
1. Reducing Local Data Copy. The first method is to reduce unnecessary local data copy which consumes CPU, *e.g.,* the two copies shown in Figure 2(a). In fact, recent work on data center networking has shown that reducing local data copies effectively lowers server CPU usage and improves throughput for Gigabit Ethernet [16, 31]. Similarly, we explore if reducing local data copy achieves benefits for smartphone networks. We present preliminary results in §4.

2. Offloading TCP/IP. When it comes to the transport protocol, the common wisdom is to modify the existing design, *e.g.,* replacing TCP with UDP, or come up with a completely new protocol, *e.g.,* QUIC [3]. These approaches, however, require server-side support and the same multi-layer stack processing (from transport, IP to link layer). Instead, we consider an alternative that does not require any server-side support — offloading TCP/IP protocol tasks to a (trusted) wireless access point or basestation. TCP offload [26] has been proposed for high speed connections like Gigabit Ethernet, where processing overhead of the network stack becomes significant. In our work, we seek to understand whether TCP offload applied to smartphones can effectively reduce CPU usage and improve both energy efficiency and performance. Our initial results are in §5.

One can further improve CPU efficiency by eliminating packetization or optimizing the driver. These, however, require advanced NIC hardware which is not yet available for smartphones. Thus we do not consider them in this study.

4. ONE-COPY

We now study the first method to reduce CPU usage by reducing local data copy. Consider an application that downloads a file and stores it in the local storage. Today, this is done by getting the data from the TCP socket buffer (*e.g.,*



(a) CPU Energy saving of 1copy (WiFi) (b) CPU Energy saving of 1copy (loopback) (c) CPU Power saving of 1copy (loopback)

Figure 3: Initial results on the benefits of one-copy using the WiFi and loopback experiments.

via *recv()*) and then writing the data to a local file descriptor (*e.g.,* via *write()*). This incurs two rounds of data copy, one from the kernel's network module to the application buffer, the other from the application buffer to the file system.

However, most file transfer and streaming applications do not modify the data. In this case, copying the same data twice is unnecessary. Instead, we let the OS directly copy the data from the TCP socket buffer to the local file buffer, or vice versa (Figure 2(b)). This bypasses the application buffer, eliminating one local data copy. Note that ideally, the residual data copy (from file to socket) can also be eliminated, *i.e.* via zero-copy [31]. This, however, requires NIC hardware support, which is not yet available for smartphones.

Implementation. Linux supports one-copy via two system calls: *sendfile()* and *splice()*. *sendfile* has been supported by Android since version 1.0 while *splice* has recently been ported to Android 5.0. One can enable one-copy in Android by replacing the TCP socket *send()* and *recv()* calls with *sendfile()* and *splice()*, respectively. No infrastructure support or special hardware is needed. We follow the same procedure to modify our networking applications mentioned in §2. For the S3 phones we experiment with both downlink and uplink but for the Note phones we only experiment with uplink (since Android 4.3 does not support *splice()*).

4.1 Initial Evaluation

We compare the two-copy and one-copy implementations under the same WiFi network conditions (same location, signal strength and no background traffic). Because S3's WiFi link rate is bounded by 24Mbps, we also run both implementations using the smartphone loopback interface to emulate the performance under higher wireless rates. We examine the *normalized CPU energy saving*: $1 - \frac{E_{CPU}(1copy)}{E_{CPU}(2copy)}$ and the absolute power saving in watt, $P_{CPU}(2copy) - P_{CPU}(1copy)$.

WiFi Results. Figure 3(a) plots the normalized energy saving for CBR streaming using S3 at various streaming rates (the result of Note is similar and thus omitted). The energy saving rate is small (<10%). The results for best effort file transfer are similar. We think this is because the WiFi link rate (<24Mbps) is much lower than the memory copy rate, the majority of CPU cycles are spent on protocol stack processing rather than local data copy.

Loopback Results. With new RF technologies, wireless link capacity is constantly growing. The newly available 802.11ac radios can achieve 1.7Gbps and the upcoming 60GHz chip will offer up to 6.7Gbps per link at a similar energy cost [5]. This trend motivates us to examine if one-copy will become (more) beneficial at higher wireless link rates.

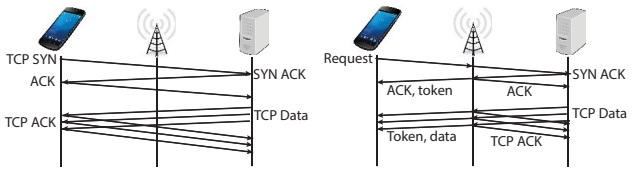
Since these new technologies are not yet ported to smartphones, we emulate a high-speed network connection using the smartphone's loopback interface, *i.e.* a local client program connects to a local server program listening on the loopback interface. We validate via measurements that the CPU overhead of the loopback experiments is almost identical to that of the WiFi experiments minus the CPU overhead of the WiFi driver (resulted omitted for brevity). Therefore, we use the CPU energy saving (by one-copy) seen from the loopback experiments to estimate those obtained from future high-speed WiFi experiments. Furthermore, since CPU is the only power consuming component, we measure the energy consumption using the Monsoon power meter. We also use it to validate the CPU energy model (see Appendix).

Figure 3(b) plots the normalized CPU energy saving of one-copy for CBR streaming. As the streaming rate increases, the overhead of memory copy becomes more significant, and one-copy achieves up to 40% CPU energy reduction over two-copy. Figure 3(c) plots the absolute CPU power saving which increases with the streaming rate. The estimated power saving can reach 100mW at 100Mbps, and 300mW at 240Mbps. For comparison, today's WiFi radios in general consume no more than 900mW power [27], and the upcoming 60GHz radios can transmit at more than 1Gbps with 700mW power [5]. Thus without any hardware change, the estimated 100-300mW power saving is significant.

5. TCP OFFLOAD

The second approach for reducing CPU load is to eliminate TCP/IP stack processing by offloading (or relocating) the process to the associated AP. This approach is complementary to the one-copy approach.

Offloading the entire TCP/IP stack to a (trustable) AP is feasible since for most of today's networks, mobile clients communicate directly with the associated AP. In home and enterprise environments, *i.e.* private networks, these APs are trustable. Consider the architecture shown in Figure 4. The smartphone OS only implements a very thin layer 2 network stack. At the start of a flow, the smartphone sends a request of TCP connection to the AP. The AP acts as a proxy, estab-



(a) Common TCP workflow (b) Stack offloading workflow

Figure 4: The architecture of TCP/IP stack offloading.

lishes a TCP connection with the remote device, and sends back a token to the smartphone. Using this token, the smartphone can send/receive packets to/from the AP who then forwards the data to the other end. No modification to the remote device is needed.

This design requires modifications at the AP, supporting two modes: TCP offload and normal TCP (for backward compatibility). It requires a new set of socket APIs on the smartphone OS, *e.g.,* a wrapping of the raw socket interfaces. The overhead for initializing a TCP offload session is just a pair of handshake packets between the smartphone and the AP (byte-level), which is negligible for large flows (over megabytes), *e.g.,* file transfer. In this paper we only consider large flows and leave the run-time decision on whether to turn on TCP offload to future work.

Implementation. We re-implement the two networking apps (file transfer and CBR streaming) using layer 2 raw sockets, bypassing TCP/IP stack. Using raw sockets, we append data directly after the Ethernet header, with the AP MAC address as the destination. We implement an AP by configuring a standard Macbook Pro (Intel Core 2 Duo 2.26GHz CPU, Broadcom BCM43224 NIC) as a WiFi hotspot and connect it to the Internet using Ethernet. We implement a proxy program on the laptop, receiving/sending layer 2 packets from/to the smartphone, and forwards data to/from a remote TCP server. Using *tcpdump* traces, we verify that our implementation does not introduce any packet loss.

For simplicity, our current design assumes that the Ethernet bandwidth is larger than the wireless bandwidth. In practice, AP's Ethernet connection rate can drop below the wireless link rate. This can be easily handled by running simple link layer flow control mechanisms, *e.g.,* explicitly control flows by either reserving bits in data frame like XON/XOFF or alerting channel conditions like RTS/CTS, and applying buffering to absorb transient bursts.

5.1 Initial Evaluation

Using both the WiFi and loopback interfaces, we study the benefits of TCP offload. As our results are consistent, we only show the S3 uplink results for brevity.

CBR WiFi Results. We consider the CBR streaming case where the same amount of data packets is transmitted under TCP offload and normal TCP. Figure 5(a) and 5(b) plot the normalized CPU energy saving and the absolute power saving of TCP offload (against normal TCP), respectively. We see that the energy benefits of TCP offload scales gracefully with the streaming rate. When the transmission rate goes beyond 20Mbps, offloading TCP leads to 40% CPU energy reduction, and 400mW power reduction.

We also compare TCP offload to UDP (for the wireless hop), which simplifies the smartphone stack processing but requires the same AP-as-proxy architecture or remote server support. With the current WiFi data rates, UDP achieves the similar energy efficiency. However, as we will show next, when the wireless data rate increases further, UDP becomes much less efficient than TCP offload.

Loopback Results. We repeat the above experiments using the loopback interface. Figure 5(c) and 5(d) report the normalized CPU energy saving and the absolute power saving against normal TCP. Surprisingly, at 40Mbps and higher, UDP's CPU energy cost exceeds that of normal TCP. This is because, as a datagram protocol, common UDP implementations maintain packet boundaries during transmissions and thus require multiple system calls per packet going through layer 4 to layer 2 processing. In contrast, TCP (and TCP offload) are streaming-based. By actively batching operations across different layers, they become much more *CPU-efficient* at higher throughput. We confirm this by increasing the MTU of the loopback interface and the UDP packet size to 20KB (compared with 1.5KB) and observing significantly higher energy efficiency (Figure 5(c) and 5(d)). Unfortunately, in practice, such large MTU configuration is not supported by WiFi NICs and APs.

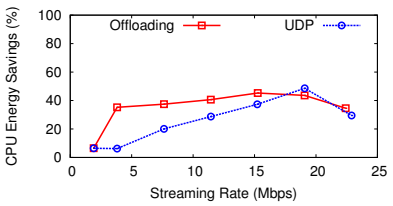
More importantly, compared with both UDP and normal TCP, TCP offload is consistently the best. When the throughput is more than 100Mbps, it saves more than 60% of CPU energy or 600mW power over normal TCP.

Best-effort WiFi Results. In addition to reducing CPU processing, TCP offload also improves the utilization of wireless links by removing TCP ACKs. To quantify this benefit (and the related energy saving), we compare TCP offload and normal TCP by running the best effort file transfer under various WiFi signal strengths. Figure 6(a) shows that TCP offload always outperforms normal TCP, achieving as much as 40% throughput improvement. Such throughput improvement also translates into energy saving, *i.e.* file transfer finishes faster at higher throughput. Figure 6(b) shows that the normalized *overall* energy saving (CPU + NIC) scales linearly with the normalized throughput gain, *e.g.,* a 40% throughput gain translates into 30% energy saving.

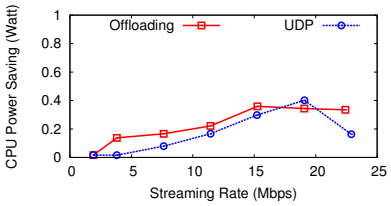
6. RELATED WORK

Optimizing OS Stacks. Reducing CPU copies has been proposed within the traditional OS community [10, 22, 28, 30]. Recent works demonstrate the necessity of CPU reduction [16, 19, 31, 33] and TCP offload [26] to improve bandwidth of the Gigabit Ethernet. Our work differs by applying CPU reduction and TCP offload techniques to smartphones. To the best of our knowledge, we are the first to demonstrate its benefits in mobile systems from an energy perspective.

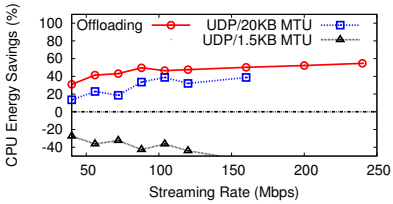
Transport Protocol Design. To improve bandwidth, many have proposed to modify or replace TCP protocols for mo-



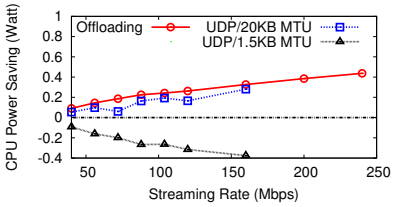
(a) Energy savings, WiFi interface



(b) Power savings, WiFi interface

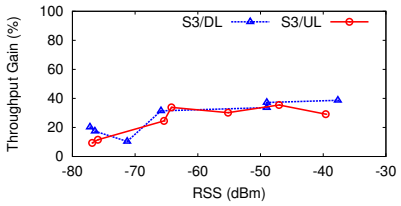


(c) Energy savings, loopback interface

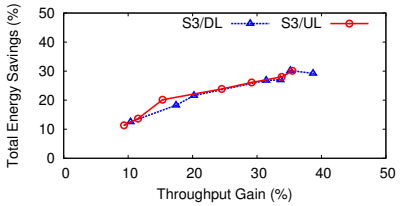


(d) Power savings, loopback interface

Figure 5: TCP offload for CBR streaming, uplink direction, S3 phones. (a) and (b) are from componentized model. (c) and (d) are actual measurements.



(a) Throughput improvement of WiFi



(b) *Total* energy ratio vs. throughput ratio

Figure 6: TCP offload for best effort file transfer, S3 phones.

bile networks, *e.g.,* the recent UDP-like QUIC protocol [3], while others everage cooperative AP/basestation to improve network performance [6, 13]. Our work differs by completely moving the TCP/IP stack out of mobile devices to improve their energy efficiency. Our approach can potentially be combined with these existing works to further improve networking performance.

Mobile Energy Saving. For software-based solutions, many have designed energy-aware adaptation techniques [2, 4, 7, 14, 20, 21, 25], and mobile cloud computing that offloads intensive computing tasks to the remote server (*e.g.,* [11, 12, 18]). Examples of hardware-driven solutions include energy harvest [17], downclocking [23, 24], and OS-level power management [9, 32]. Our work differs by exploring a new and complementary dimension, *i.e.* trimming the network stack, which effectively reduces CPU energy cost.

7. SUMMARY AND FUTURE WORK

Our proposals for reducing CPU usage in networking operations have produced promising initial results. Using today's WiFi configurations (<24Mbps), our techniques achieve moderate energy savings, *i.e.* 10% for one-copy, and 40% for TCP offload. And as wireless link capacity increases (>100Mbps), the estimated improvements become much more substantial, *e.g.,* 40% for one-copy and 60% for TCP offload.

Following this first step, more work is required to explore and compare methods of CPU usage reduction. Here we discuss possible deployment challenges of our techniques, and open research problems related to our work.

- *Practical Deployments:* Because one-copy bypasses application buffers, applications can no longer access/modify data during transmission. This might require decoupling data computation from data transmission in some streaming video applications or encrypted communications. TCP offload requires a cooperative AP or basestation, which is feasible for home or enterprise WiFi APs who seek improved performance. For cellular providers, TCP offload can be offered as a premium service to customers willing to pay for better energy efficiency and performance.

- *Handling Packet Losses:* TCP offload assumes reliable transmissions on the last hop to the mobile device. Modern MAC protocols like 802.11n/ac already implement ARQ and link adaptation mechanisms to minimize and recover from wireless losses. Upon persistent failures (*e.g.,* hardware malfunction), link layer losses will propagate to the application layer. To minimize such impact, we need link layer designs that track packet losses and notify the application layer.

- *Offload Mode Selection:* TCP offload is not suitable to all scenarios. Smartphones need to decide whether to offload network stack based on AP/basestation availability, flow properties (*e.g.,* large vs. very small flows), and channel conditions (low vs. high loss). We need to build and integrate such management function into smartphone OS and evaluate its performance under diverse real-life conditions.

- *Efficient AP Design:* TCP offload introduces extra processing overhead at the AP. At 24Mbps, such processing takes no more than 5% CPU utilization on our 6-year-old laptop. There is still large room for improvement: currently we use the traditional *libpcap* to receive and forward packets; by switching to high performance packet processing engines like *DPDK* with further optimizations, the AP processing efficiency should be at least doubled.

- *Applications to Wearables and IoT:* Wearable devices have higher requirements for energy efficiency. We can apply similar methods to trim their network stacks, *e.g.,* offload TCP to associated smartphones. Here, application and flow properties can be very different, *e.g.,* more short flows, but more stable connections so lower overhead from token requests. A unique challenge here is that we must be aware of the impact of TCP offloading on the smartphone energy level. Balancing energy consumption of smartphones and wearables is an interesting open question.

8. REFERENCES

[1] https://help.netflix.com/en/node/306.

[2] Optimizing for doze and app standby. http://developer.android.com/training/monitoring-device-state/doze-standby.html.

[3] Quic, a multiplexed stream transport over udp. https://www.chromium.org/quic.

[4] AGGARWAL, B., CHITNIS, P., DEY, A., JAIN, K., NAVDA, V., PADMANABHAN, V. N., RAMJEE, R., SCHULMAN, A., AND SPRING, N. Stratus: Energy-efficient mobile communication using cloud support. In *Proc. of SIGCOMM* (2010).

[5] ANANDTECH. MWC 2014: Wilocity brings WiGig 802.11ad to smartphones with wil6300 chipset.

[6] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., AND KATZ, R. H. A comparison of mechanisms for improving tcp performance over wireless links. In *Proc. of SIGCOMM* (1996).

[7] BANERJEE, N., RAHMATI, A., CORNER, M. D., ROLLINS, S., AND ZHONG, L. Users and batteries: Interactions and adaptive energy management in mobile systems. In *Proc. of UbiComp* (2007).

[8] CHEN, X., DING, N., JINDAL, A., HU, Y. C., GUPTA, M., AND VANNITHAMBY, R. Smartphone energy drain in the wild: Analysis and implications. In *Proc. of SIGMETRICS* (2015).

[9] CHEN, X., JINDAL, A., DING, N., HU, Y. C., GUPTA, M., AND VANNITHAMBY, R. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proc. of MobiCom* (2015).

[10] CHU, H. J. Zero-copy tcp in solaris. In *Proc. of USENIX ATC* (1996), pp. 21–21.

[11] CUERVO, E., WOLMAN, A., COX, L. P., LEBECK, K., RAZEEN, A., SAROIU, S., AND MUSUVATHI, M. Kahawai: High-quality mobile gaming using GPU offload. In *Proc. of MobiSys* (2014).

[12] DINH, H. T., LEE, C., NIYATO, D., AND WANG, P. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing 13*, 18 (2013), 1587–1611.

[13] FARKAS, V., HÃL'DER, B., AND NOVÃĄCZKI, S. A split connection tcp proxy in lte networks. In *Information and Communication Technologies*. 2012.

[14] FONSECA, R., AND MARTINS, M. Application modes: A narrow interface for end-user power management in mobile devices. In *Proc. of HotMobile* (2013).

[15] GOPAL, S., AND RAYCHAUDHURI, D. Investigation of the tcp simultaneous-send problem in 802.11 wireless local area networks. In *Proc. of ICC* (2005).

[16] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. of NSDI* (2014).

[17] KELLOGG, B., PARKS, A., GOLLAKOTA, S., SMITH, J. R., AND WETHERALL, D. Wi-fi backscatter: Internet connectivity for RF-powered devices. In *Proc. of SIGCOMM* (2014).

[18] KEMP, R., PALMER, N., KIELMANN, T., AND BAL, H. Cuckoo: a computation offloading framework for smartphones. In *Proc. of MobiCASE* (2010).

[19] KURMANN, C., RAUCH, F., AND STRICKER, T. M. Speculative defragmentation - leading gigabit ethernet to true zero-copy communication. *Cluster Computing 4*, 1 (2001), 7–18.

[20] LEE, K., RHEE, I., LEE, J., CHONG, S., AND YI, Y. Mobile data offloading: how much can wifi deliver? In *Proc. of CoNEXT* (2010).

[21] LI, C., PENG, C., LU, S., AND WANG, X. Energy-based rate adaptation for 802.11n. In *Proc. of MobiCom* (2012).

[22] LIU, T., ZHU, H., CHANG, G., AND ZHOU, C. The design and implementation of zero-copy for linux. In *Proc. of ISDA* (2008).

[23] LU, F., LING, P., VOELKER, G. M., AND SNOEREN, A. C. Enfold: Downclocking OFDM in WiFi. In *Proc. of MobiCom* (2014).

[24] LU, F., VOELKER, G. M., AND SNOEREN, A. C. SloMo: Downclocking wifi communication. In *Proc. of NSDI* (2013).

[25] MAITI, A., CHEN, Y., AND CHALLEN, G. Jouler: A policy framework enabling effective and flexible smartphone energy management. In *Proc. of MobiCASE* (2014).

[26] MOGUL, J. C. TCP offload is a dumb idea whose time has come. In *Proc. of HotOS* (2003).

[27] NIKA, A., ZHU, Y., DING, N., JINDAL, A., HU, Y. C., ZHOU, X., ZHAO, B. Y., AND ZHENG, H. Energy and performance of smartphone radio bundling in outdoor environments. In *Proc. of WWW* (2015).

[28] O'CARROLL, F., TEZUKA, H., HORI, A., AND ISHIKAWA, Y. The design and implementation of zero copy MPI using commodity hardware with a high performance network. In *Proc. of ICS* (1998).

[29] PATHAK, ABHINAV, Y. C. H., AND ZHANG, M. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proc. of EuroSys* (2012).

[30] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *Proc. of OSDI* (2014).

[31] RIZZO, L. Netmap: A novel framework for fast packet I/O. In *Proc. of USENIX ATC* (2012).

[32] XU, C., LIN, F. X., WANG, Y., AND ZHONG, L. Automated os-level device runtime power management. In *Proc. of ASPLOS* (2015).

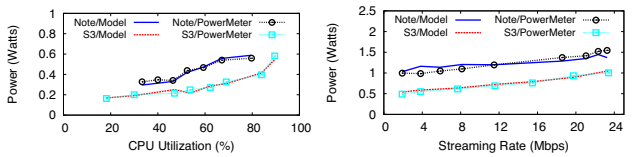
[33] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale RDMA deployments. In *Proc. of SIGCOMM* (2015).

Acknowledgment

We would like to thank Charlie Y. Hu for his suggestions on the power monitor validation. We also thank anonymous reviewers for their useful feedback. This work is supported by NSF grants AST-1443956, AST-1443945 and CNS-1224100. Any opinions, findings, and conclusions or recommendations expressed in this material do not necessarily reflect the views of any funding agencies.

Appendix: Validating Energy Models

We validate the energy models using power meter measurements. *First*, we validate the CPU energy model by running a bare-bone application that drives the CPU to different utilization levels, recording both the CPU utilization (which drives the model) and the power meter readings. Figure 7(a) shows that the model faces less than 10% error compared with power meter measurements. This result aligns with [27]. Experiments on loopback follow a similar result since it bypasses NIC and consumes CPU power only. *Next*, we validate the WiFi NIC power model. We run WiFi transmissions at different CBR rates, and measure the total energy consumption using the power meter. We then predict the CPU and NIC energy consumption using the CPU and NIC power models, and compare the sum to the power meter readings. Figure 7(b) shows that the predicted sum aligns with the power meter results. Together, these experiments validate the componentized power models.



(a) CPU Power (b) WiFi + CPU Power

Figure 7: Validating the componentized energy models by comparing them to the power meter readings.