Energy-Aware CPU Frequency Scaling for Mobile Video Streaming

Yi Yang, *Student Member, IEEE,* Wenjie Hu, *Student Member, IEEE,* Xianda Chen, *Student Member, IEEE,* Guohong Cao, *Fellow, IEEE,*

Abstract—The energy consumed by video streaming includes the energy consumed for data transmission and CPU processing, which are both affected by the CPU frequency. High CPU frequency can reduce the data transmission time but it consumes more CPU energy. Low CPU frequency reduces the CPU energy but increases the data transmission time and then increases the energy consumption. In this paper, we aim to reduce the total energy of mobile video streaming by adaptively adjusting the CPU frequency. Based on real measurement results, we model the effects of CPU frequency on TCP throughput and system power. Based on these models, we propose an Energy-aware CPU Frequency Scaling (EFS) algorithm which selects the CPU frequency that can achieve a balance between saving the data transmission energy and CPU energy. Since the downloading schedule of existing video streaming apps is not optimized in terms of energy, we also propose a method to determine when and how much data to download. Through trace-driven simulations and real measurement, we demonstrate that the EFS algorithm can reduce 30% of energy for the Youtube app, and the combination of our download method and EFS algorithm can save 50% of energy than the default Youtube app.

Index Terms—Energy efficiency, video streaming, cellular networks, smartphone

1 Introduction

Video streaming has become extremely popular on mobile devices over the last few years. Mobile video streaming on Youtube, Netflix, has taken 55% of the total mobile data traffic in 2015, and will take 75% by 2020 [1]. Since video has much larger data size, a large amount of energy will be consumed to download video on smartphones. Thus, it is critical to improve the energy efficiency of video streaming on smartphones.

The energy consumption of video streaming includes the energy consumed for data transmission and the energy consumed for CPU processing such as decoding. The data transmission energy itself includes the wireless interface energy and the CPU energy consumed to process packets. To reduce the data transmission energy, a widely used method is to download some amount of video content as fast as possible and then turn the wireless interface off [2], [3], [4]. Since the CPU energy is related to its working frequency [5], [6], it is possible to reduce the CPU energy by decreasing its frequency.

A straightforward method to save the energy consumption of video streaming during data transmission is to reduce the wireless interface energy and the CPU energy separately. However, these two goals are contradictory because the TCP throughput is related to the CPU frequency [7]. High CPU frequency can help increasing the TCP throughput and thus saving the wireless interface energy by reducing the data transmission time, but it costs much more CPU energy. On the other hand, low CPU frequency reduces the CPU energy, but makes the CPU a bottleneck and affects the TCP throughput. It increases the data transmission time and thus increase the wireless interface energy. To reduce the total energy of video streaming during data transmission, the CPU frequency should be properly setup to achieve a balance between the wireless interface energy and the CPU energy.

For modern smartphones, the CPU can work at a series of frequencies. For example, Samsung Galaxy S4 with Cortex-A15 can work at 10 different frequencies, and Samsung Galaxy S5 using Qualcomm Snapdragon 801 can work at 15 different frequencies. The CPU frequency and the voltage provided to the CPU can be adjusted at runtime. This feature is called Dynamic Voltage and Frequency Scaling (DVFS). The system driver uses different policies to adjust the CPU frequency, which are called the *CPU governors*. For instance, the default CPU governor used by most smartphones is the *interactive* governor, which adjusts the CPU frequency according to the CPU usage. However, the default CPU governor tends to set the CPU at high frequency to provide better performance, which consumes a large amount of energy. Other CPU governors, such as the *powersave* governor, can restrict the CPU frequency to a low value, but they may increase the data transmission time and energy.

In this paper, we aim to reduce the total energy of mobile video streaming by properly adjusting the CPU frequency. Based on real measurement results, we find that the CPU may become a bottleneck and affect the TCP throughput when its frequency is low, and then we model the effects of CPU frequency on TCP throughput and power consumption. Based on these models, we propose an Energy-aware CPU Frequency Scaling (EFS) algorithm which selects the CPU frequency that can achieve a balance between

• *Yi Yang, Wenjie Hu, Xianda Chen and Guohong Cao are with the Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802. E-mail: {yzy123, wwh5068, xuc23, gcao}@cse.psu.edu*

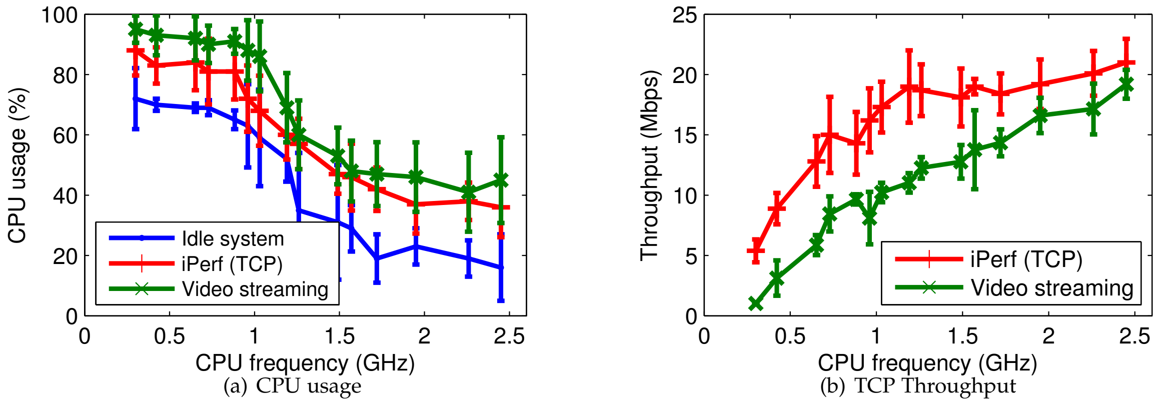


Fig. 1. The impact of CPU frequency on TCP throughput. When the CPU usage is higher than 70%, CPU becomes a bottleneck and affects TCP throughput (Samsung Galaxy S5).

data transmission energy and CPU energy. Since the downloading schedule of existing video streaming apps is not optimized in terms of energy, we also propose a method to determine when and how much data to download. The efficiency of EFS algorithm and our downloading method is verified by trace-driven simulations and real measurement. Evaluation results show that the EFS algorithm can reduce 30% of energy, and the combination of our download method and EFS algorithm can save 50% of energy, when compared to the Youtube app.

The contribution of this paper can be summarized as follows.

- We are the first to study the relationship between TCP throughput, system power and CPU frequency in video streaming. Based on measurement results, we model the effects of CPU frequency on TCP throughput and system power.
- We propose an Energy-aware CPU Frequency Scaling (EFS) algorithm to reduce the total energy for video streaming. During data transmission, EFS selects the most energy efficient CPU frequency considering both CPU energy and data transmission energy. When there is no data transmission, EFS selects a low CPU frequency to reduce the energy consumption without affecting the user experience. We further propose to integrate our EFS algorithm with DASH considering unstable network condition.
- We consider the impact of the downloading schedule on energy and combine it with our EFS algorithm to further improve the energy efficiency of video streaming.

The rest of this paper is organized as follows. Section 2 introduces the background and our motivation to save energy using CPU frequency scaling. Section 3 presents the EFS algorithm and Section 4 presents our energy efficient downloading schedule for video streaming. The evaluation results are shown in Section 5. Section 6 introduces the related work. Section 7 concludes the paper.

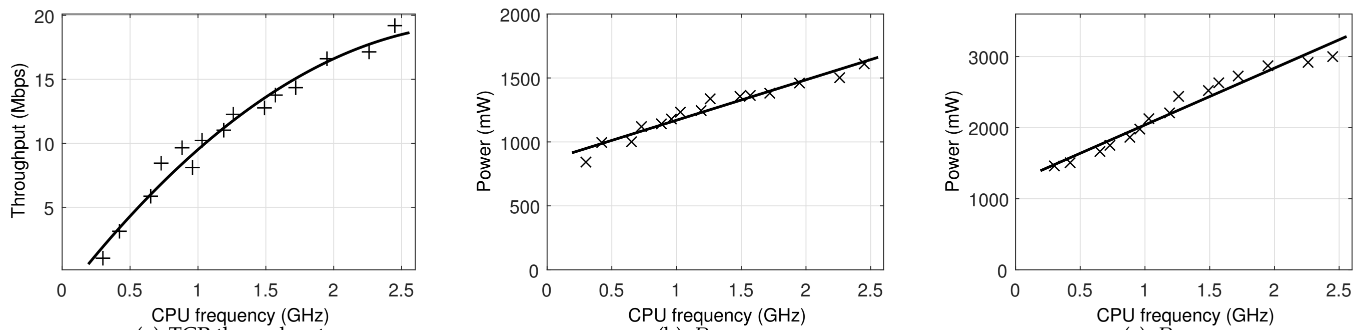
2 Background and Motivation

In this section, based on real measurement results, we model the effects of CPU frequency on TCP throughput and system power. Then we introduce our motivation to save energy by adjusting CPU frequency.

2.1 Measurement Setup

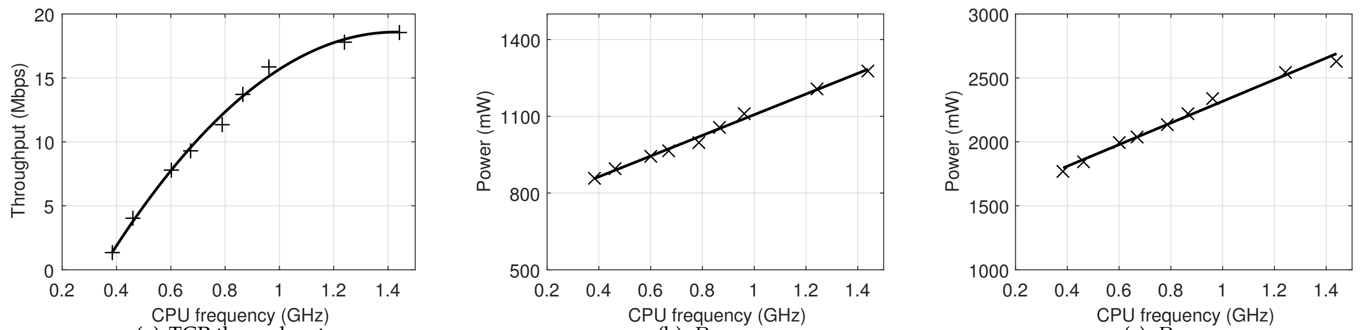
To model the impact of CPU frequency on TCP throughput and system power, we collect real measurement data related to TCP throughput and power consumption under different CPU frequencies. Our testbed includes a rooted Samsung Galaxy S5 and a rooted LG Nexus 5x. Samsung Galaxy S5 is equipped with Qualcomm Snapdragon 801 CPU, which can work at 15 different frequencies from 300 MHz to 2.45 GHz. During the measurement, the CPU frequency was tuned on CPU core 0. LG Nexus 5x is equipped with two types of CPU cores, which are Cortex-A53 and Cortex-A57. Cortex-A53 is designed for high energy efficiency, which is used for running normal tasks and can work on 9 different frequencies from 384 MHz to 1.44 GHz. Cortex-A57 is designed for high computation performance, which is usually turned off and only being activated for computationally intensive tasks. We tested the Youtube app to watch videos with various resolutions and found that CPU core 0 (Cortex-A53) was running while Cortex-A57 was always kept off during video streaming. Thus, CPU core 0 was considered in our testbed. We used the 3C CPU Manager [8] to set the CPU working at a specific frequency and used the OS monitor to record the real time CPU usage.

The TCP throughput measurement was based on AT&T's LTE network. We used the Youtube app to watch a video with constant bit rate (720p) for 1 minute at different CPU frequencies. At each CPU frequency, we collected the network trace using TCPDUMP, which recorded the timestamp and data size of each packet. All packets with an interval less than 1 second were considered as one downloading period, and we computed the TCP throughput as the average value among all the downloading periods. Since the CPU can work at 15 frequencies, running a set of tests took around 20 minutes. The network condition may vary within this time period. To reduce the effects of network fluctuation, our measurements were done at night when there were fewer users in the LTE network. Also, we measured the TCP throughput at different frequencies alternatively. For example, we changed the CPU frequency



(a) TCP throughput (b) P_{idle} (c) P_{tran}

Fig. 2. The TCP throughput and power model considering CPU frequency on CPU core 0 (Samsung Galaxy S5)



(a) TCP throughput (b) P_{idle} (c) P_{tran}

Fig. 3. The TCP throughput and power model considering CPU frequency on CPU core 0 (LG Nexus 5x)

from 300 MHz to 2.45 GHz, and then from 2.45 GHz to 300 MHz again on Samsung Galaxy S5. This process was repeated for five times to form one group of tests and the tests were run for 10 nights to obtain the average TCP throughput at different CPU frequencies. To measure the power consumption, we used the Monsoon power monitor to provide power directly to the smartphones, which can record the power value at a sample rate of 5000 Hz.

2.2 Impact of CPU on TCP Throughput

CPU usage is the percentage of CPU time used to process instructions, other than waiting. It is used to describe the load of the CPU. When the CPU usage is above 70%, it may become a bottleneck and affect the user experience. For smartphones, the operating system itself consumes a large amount of CPU. Fig. 1 shows the experimental results for Samsung Galaxy S5. In Fig. 1(a), we show the CPU usage of idle system (all user applications are turned off). As can be seen, when the CPU frequency decreases, the CPU usage increases, which may affect the performance of user applications. Video streaming uses TCP as the transport layer protocol, and TCP uses lots of CPU capacity to handle congestion avoidance issues, buffer and reorder received packets, request the retransmission of missing packets, etc. On top of TCP, video streaming has complex application layer operations, such as moving data from the TCP buffer to the application buffer, decoding the received content and displaying them on screen, and thus requires more CPU capacity.

When the CPU frequency is low, the remaining CPU capacity may not be enough to process the TCP task and video streaming, and thus affecting the TCP throughput. To verify this, we measure the CPU usage and TCP throughput of two apps: iPerf (without application layer operation) and Youtube. As shown in Fig. 1, the CPU usage increases when its frequency decreases, and the TCP throughput decreases accordingly for both iPerf and Youtube. Also, the TCP throughput of Youtube is lower than that of iPerf and is more sensitive to the change of CPU frequency, as it has application layer operations. We also notice that the TCP throughput of iPerf increases quickly when the CPU frequency increases from 0.3GHz to 1.19GHz, but becomes almost flat after that, where the CPU is not a bottleneck. However, video streaming requires much more CPU capacity and hence the impact of CPU frequency on throughput is much higher.

2.3 TCP Throughput and Power Model

Here we mainly consider the TCP throughput of video streaming. The average value of TCP throughput at different CPU frequency is drawn in Fig. 2(a) for Samsung Galaxy S5 and Fig. 3(a) for LG Nexus 5x, and the TCP throughput $r(f)$ can be described as $r(f) = r_{max} \times r^*(f)$. The first part r_{max} is the network throughput which is not related to the variation of CPU frequency, and is only affected by the signal strength, location, the number of users nearby, etc. r_{max} may vary from time to time, however, during a short time period when watching a video (within 10 minutes in most cases

TABLE 1
Power model considering CPU frequency

	State	Power (mW)	Duration (s)
Samsung Galaxy S5	Idle	$P_{idle}(f) = 315.7f + 854$	-
	Promotion	$P_{pro}(f) = 639.2f + 1206$	$t_{pro} = 0.91$
	Data trans.	$P_{tran}(f) = 799.1f + 1241$	-
	Tail	$P_{tail}(f) = 288.3f + 1119$	$t_{tail} = 10.35$
LG Nexus 5x	Idle	$P_{idle}(f) = 404.2f + 702$	-
	Promotion	$P_{pro}(f) = 639.2f + 1206$	$t_{pro} = 0.91$
	Data trans.	$P_{tran}(f) = 846.2f + 1471$	-
	Tail	$P_{tail}(f) = 288.3f + 1119$	$t_{tail} = 10.35$

[9]), it is relatively stable. In the experiment, we measure r_{max} under a stable network environment and model it using the average network throughput. More accurate measurement of network throughput can be found in [10], [11], which is out of the scope of this paper. The second part $r^*(f)$ describes the impact of CPU frequency on TCP throughput, which may vary with different phone models and the relationship can be modeled and trained offline. Since it only needs to be trained once, it does not introduce much overhead. To be more accurate, the model may rely on other parameters such as locations (i.e., home or office). For example, the TCP throughput of our testbed at office is described as $r(f) = 19.19 \times (-0.12 \times f^2 + 0.71 \times f - 0.1)$ for Samsung Galaxy S5 and $r(f) = 18.56 \times (-0.85 \times f^2 + 2.43 \times f - 0.73)$ for LG Nexus 5x.

In LTE, the wireless interface can work in four states: *idle*, *promotion*, *data transmission* and *tail*. Initially the LTE interface is in the idle state when there is no data transmission. When a data transmission request comes, it enters the promotion state to negotiate with the base station to obtain the data transmission channels. Then it transfers to the data transmission state and begins to transmit data. After data transmission, the LTE interface can not go back to idle directly. It is forced to stay in the tail state and wait for about 10 seconds before going to the idle state. During the tail state, the phone still holds the data transmission channel, and can serve the next data transmission request immediately.

The power consumption of a smartphone is related to both the cellular interface and the CPU frequency. Since LTE has four states, we build four power models correspondingly. Our models describe the whole phone's power and use CPU frequency as an important parameter. We first model the idle power ($P_{idle}(f)$) when there is no data transmission. As the video is decoded and played, we measure $P_{idle}(f)$ when the phone plays a piece of locally cached video, downloaded from Youtube. The power consumption in idle state as a function of CPU frequency is drawn in Fig. 2(b) for Samsung Galaxy S5 and Fig. 3(b) for LG Nexus 5x. As can be seen, it generally increases linearly with the CPU frequency. Note that in some previous work [6], the CPU power consumption increases super linearly with the frequency, since they only consider the power of CPU, where the voltage also changes linearly with the CPU frequency. Different from them, we consider the power consumption of the whole smartphones, where the voltage provided by the battery is a constant number. As a result, the power consumption changes linearly with the CPU frequency, similar to [5]. We also measure the power consumption in other states by watching videos using the Youtube app. Similar to the idle state, the power consumption in these three states also has linear relationship with the CPU frequency, but with different slops (parameters). For example, the relationship between $P_{tran}(f)$ and the CPU frequency is shown in Fig. 2(c) for Samsung Galaxy S5 and Fig. 3(c) for LG Nexus 5x. For our testbed, the power models of different states are summarized in Table 1, where f is in GHz and power is in mW.

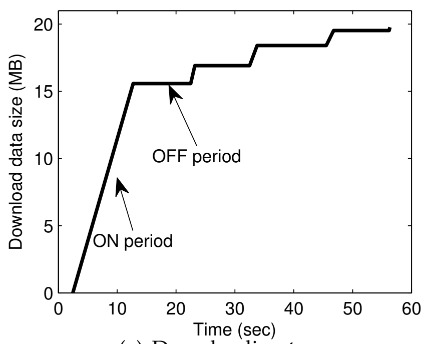
2.4 Motivation

Video streaming apps need to download video content periodically. Most existing mobile video streaming apps use ON-OFF scheme to download video. Figure 4(a) shows a downloading trace of Youtube. As can be seen, it downloads some data and then turns the wireless interface off to save energy. After a while, when the buffered content is almost run out, it triggers another round of data downloading. Thus video streaming can be seen as a set of data downloading tasks. In this paper, we aim to find the proper CPU frequency for the data downloading tasks to minimize the total energy. At first, we use an example of one task to illustrate the importance of selecting CPU frequency, as shown in Fig. 4(b). As the downloaded data can be played for a period of time, we consider the total energy includes data transmission, tail, and the idle part. Note that the idle part may not exist if the playback time is shorter than the data transmission time.

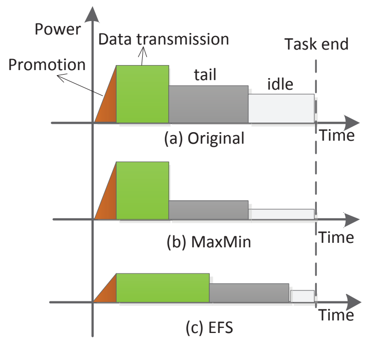
For the original system (Fig. 4(b)) which uses the default interactive CPU governor, the CPU always works at very high frequency during data transmission, and then slowly degrades to a lower frequency, which is still much higher than the requirement. Thus, it consumes a large amount of energy. To save energy, a straightforward method is to use the highest CPU frequency during data transmission and then switch to a much lower CPU frequency that is enough to play the video back after data transmission[1]. This method is referred to as the *MaxMin* method (Fig. 4(b)). Clearly it can save lots of energy when compared to the original system. However, using the highest CPU frequency during data transmission does not always consume minimum energy.

To see the relationship between minimum energy consumption and CPU frequency, we compute the total energy when downloading different amount of data at a given CPU frequency on Samsung Galaxy S5, and show the results in Fig. 4(c). The y-axis is the relative energy when compared

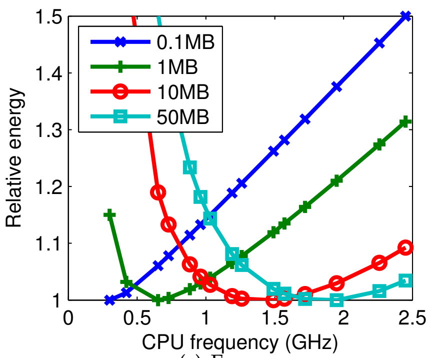
1. As the promotion state is short, we assume its CPU frequency stays the same as the data transmission state, to reduce its frequent changes.



(a) Downloading trace



(b) Comparison of different methods



(c) Energy

Fig. 4. The motivation of our method

to the minimum energy consumed to transmit the same amount of data. As can be seen from the figure, setting the CPU frequency to 2 GHz can minimize the energy when 50 MB is needed. However, setting the CPU frequency to 2 GHz will consume 22% more energy if only 1MB is needed, and it will consume 38% more energy if only 0.1 MB is needed. This is because the wireless interface works in multiple states (e.g., promotion, data transmission), and adjusting the CPU frequency not only affects the data transmission power but also affects the promotion power. If the data size is very small (i.e., 0.1 MB), the data transmission time is very short and the promotion energy accounts for a larger proportion of the total energy compared to the data transmission energy. Then although a lower CPU frequency may increase the data transmission energy, it can save more energy by reducing the promotion energy. If the data size is large, the data transmission energy dominates the total energy. Then although a higher CPU frequency may increase the promotion energy, it can save more energy by reducing the data transmission energy. Thus, different CPU frequencies should be selected for different data sizes. Based on this finding, we introduce an Energy-aware CPU Frequency Scaling (EFS) algorithm to find the optimal CPU frequency considering the task size and the network conditions. During data transmission, we will select the proper CPU frequency to reduce energy and guarantee that the data is downloaded in time. Since we use lower CPU frequency, the downloading time is longer and the idle time is shorter than the MaxMin method, and thus more energy can be saved.

We notice that the downloading schedule of existing video streaming apps (e.g., Youtube in Fig. 4(a)) is not optimized for energy. Thus, we introduce a method to optimize the downloading schedule, i.e., how much data to download and when to download. By combining the downloading schedule and EFS algorithm, we can save much more energy than the existing methods.

3 Energy-Aware CPU Frequency Scaling for Existing Video Streaming Apps

In this section, we introduce our Energy-aware CPU Frequency (EFS) algorithm to select the most energy efficient CPU frequency for existing video streaming apps.

3.1 Problem Statement

The video streaming process can be considered as a set of n data transmission tasks. Task T_i needs to download d_i data from time t_i. For an existing video streaming app, the downloading schedule is determined by the application, i.e., d_i and t_i can be considered as given value (see Fig. 4(a)). Note that for each task (chunk), the video resolution can also be different and may be adaptively adjusted based on the network throughput (r_{max} in our model). Thus our solution can also be directly applied to adaptive video streaming, such as MPEG-DASH and Adobe HTTP Dynamic Streaming.

To guarantee that the video is played smoothly, T_i must be downloaded before the next downloading period. In this paper we also call t_{i+1} as the task end of T_i. For the last task T_n, the task end is the time when the whole video is played out. The duration from the start to the end of a task is called the length of a task. The energy consumption of task T_i is defined as the total energy consumed from the start to the end, and our goal is to minimize the total energy of all the tasks, which can be described as **minimize** $\sum_{i=1}^{n} E(T_i)$.

3.2 The Energy Consumption of One Task

The energy consumption of a task contains the data transmission energy and the CPU energy. Based on the starting and finishing states of a task, the energy consumption of T_i can be computed in four cases, as illustrated in Fig. 5. In each case, we assume the CPU works at one frequency during data transmission and a lower frequency when there is no data transmission. During data transmission, we select a frequency from the CPU frequency set F to achieve a good tradeoff between reducing the data transmission energy and the CPU energy. When the data transmission is done, the CPU works at a lower frequency f_{min} which reduces the CPU energy and also provides satisfactory performance. We do not consider the case where the data transmission can not be finished before the task end, since it violates the downloading schedule and then affects user experience [12].

Case (a): As shown in Fig. 5 (a), the LTE interface is in the idle state at the beginning of T_i. Thus, it enters promotion state first and pay extra promotion energy. In this case, the CPU works at relative high frequency (f_a) and the TCP throughput is relative high ($r(f_a) > \frac{d_i}{l_i - t_{tail} - t_{pro}}$), so the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2018.2878842, IEEE Transactions on Mobile Computing

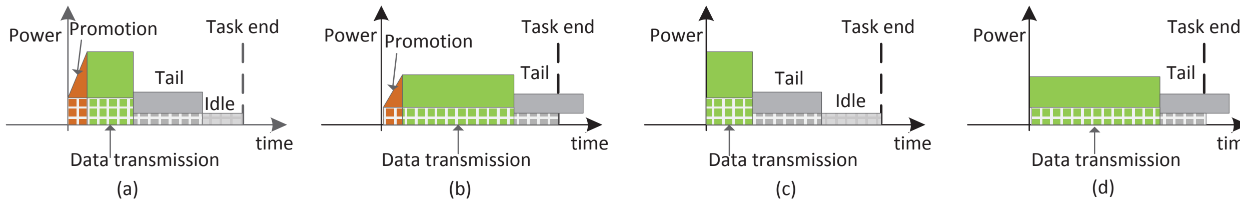


Fig. 5. The four cases to compute the energy of a task, which is defined as the total energy from the start time to the task end. The dotted filling indicates the CPU power in the corresponding state.

phone demotes to idle state at the end of the task. The total energy can be computed using Eq. 1.

$$\begin{aligned} E(T_i)^a =& t_{pro} \times P_{pro}(f_a) + \frac{d_i}{r(f_a)} \times P_{tran}(f_a) \\ &+ t_{tail} \times P_{tail}(f_{min}) \\ &+ (l_i - t_{pro} - t_{tail} - \frac{d_i}{r(f_a)}) \times P_{idle}(f_{min}), \\ &\text{if } r(f_a) > \frac{d_i}{l_i - t_{tail} - t_{pro}} \end{aligned} \tag{1}$$

Case (b): The energy consumption of Case (b) is shown in Fig. 5 (b). Similar to Case (a), the wireless interface is in the idle state at the beginning of T_i. However, the TCP throughput in this case is low as low CPU frequency is selected. At the end of task T_i, the LTE interface is still in the tail state. The total energy is computed as Eq. 2. In this case the next task can skip the promotion state and start data transmission immediately.

$$\begin{aligned} E(T_i)^b =& t_{pro} \times P_{pro}(f_b) + \frac{d_i}{r(f_b)} \times P_{tran}(f_b) \\ &+ (l_i - t_{pro} - \frac{d_i}{r(f_b)}) \times P_{tail}(f_{min}), \\ &\text{if } \frac{d_i}{l_i - t_{pro}} < r(f_b) \le \frac{d_i}{l_i - t_{tail} - t_{pro}} \end{aligned} \tag{2}$$

Case (c): As shown in Fig. 5 (c), the wireless interface is in high power state at the beginning, so data transfer starts immediately. Similar to Case (a), the TCP throughput is assumed to be high in this case ($r(f_c) > \frac{d_i}{l_i - t_{tail}}$). Then the total energy is computed as Eq. 3.

$$\begin{aligned} E(T_i)^c =& \frac{d_i}{r(f_c)} \times P_{tran}(f_c) + t_{tail} \times P_{tail}(f_{min}) \\ &+ (l_i - t_{tail} - \frac{d_i}{r(f_c)}) \times P_{idle}(f_{min}), \\ &\text{if } r(f_c) > \frac{d_i}{l_i - t_{tail}} \end{aligned} \tag{3}$$

Case (d): The energy consumption of Case (d) is shown in Fig. 5 (d), where the data transmission starts immediately, similar to Case (c). But the TCP throughput is assumed to be low and the LTE interface is still in the tail state at the task end. The total energy is computed as Eq. 4.

$$\begin{aligned} E(T_i)^d =& \frac{d_i}{r(f_d)} \times P_{tran}(f_d) \\ &+ (l_i - \frac{d_i}{r(f_d)}) \times P_{tail}(f_{min}), \\ &\text{if } \frac{d_i}{l_i} < r(f_d) \le \frac{d_i}{l_i - t_{tail}} \end{aligned} \tag{4}$$

For task T_i, we compute the energy in all of the four cases. In each case we search for the CPU frequency that can minimize the energy. Then we define the minimum energy in the four cases as the min energy of task T_i, as shown in Eq. 5.

$$\begin{aligned} E(T_i) \in \{\min E(T_i)^a, \min E(T_i)^b, \min E(T_i)^c, \min E(T_i)^d\} \\ f_a, f_b, f_c, f_d \in F \end{aligned} \tag{5}$$

3.3 Energy-Aware CPU Frequency Scaling Algorithm

For each task, it is easy to obtain the minimum energy as there are only four cases. However, since the ending of previous tasks also affects the energy of later tasks, minimizing the energy of every task individually may not minimize the total energy of all tasks. To solve this problem, we propose an energy-aware CPU frequency scaling (EFS) algorithm which aims to find the global optimal solution. Our key idea is to map this problem to the shortest path problem.

We build a directed graph as shown in Fig. 6. For each task, there are four cases to compute the energy, as illustrated in Fig. 5, except the first one, which only has two cases since the wireless interface is in the idle state at the beginning. Each energy case of a task is mapped to one node in the graph. For example, the two cases of task 1 map to nodes $1a$ and $1b$ in Fig. 6. Besides these nodes, we add a virtual start and virtual end node. Next we add links to the graph. The links between two nodes indicate a possible schedule between the two tasks. For example, if task T_i is downloaded using Case (a), then the LTE interface enters idle state at the end, so task T_{i+1} can only be scheduled by Case (a) or Case (b). Thus, we add directed links from node ia to nodes $(i+1)a$ and $(i+1)b$. The other links between task nodes are added similarly. For the two virtual nodes, we add links from the virtual start node to the two cases of task 1, and add links from the four cases of task n to the virtual end node. The weight of a link is the energy consumption

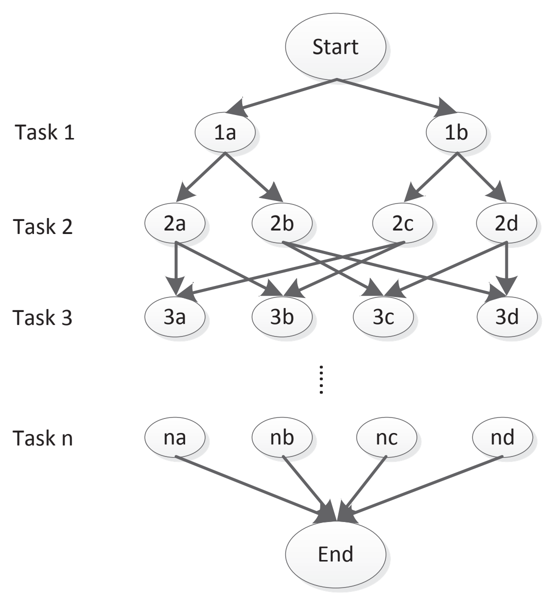


Fig. 6. Mapping the minimum energy of video streaming to the shortest path problem

of the node at the end of the link. For the four links from task n to the virtual end node, their weight is defined as 0. In this graph, we take into account all power cases of each task and all possible schedule paths between tasks, so each path from the virtual start node to the virtual end node will map to one schedule of all tasks, and vice versa. As a result, the minimum energy of all tasks corresponds to the shortest path from the virtual start node to the virtual end node.

Based on the graph we can use the Dijkstra algorithm to find out the shortest path. Given n tasks, the number of nodes in the graph is $O(4n)$, and the number of edges is $O(8n)$. As the time complexity of the shortest path algorithm is $O(V+E)logV$, where V is the number of nodes and E is the number of edges, then the time complexity is $O(n\log n)$ in our case. Additionally, as the CPU can work at $|F|$ discrete frequencies, computing the weight of each link need to consider all the $|F|$ possibilities. Putting them together, the time complexity of the EFS algorithm is $O(|F|n\log n)$.

3.4 Minimum CPU Frequency Selection

In previous sections, we assume the minimum CPU frequency without data transmission (f_{min}) is a constant value. In fact, this value is related to the video resolution. To obtain this minimum CPU frequency under a specific video resolution, one simple solution is to play videos from the local storage, and then measure the CPU frequency. However, this minimum CPU frequency would be smaller than what is needed. During video streaming, the system also needs to maintain the buffer and TCP connection even when there is no data transmission. There may be some background apps that will consume extra CPU capacity, so the system will require a higher CPU frequency.

To solve this problem, we use Youtube to stream a video at the given resolution and pause it to buffer a long period of video. Then, we tune the CPU frequency and search for the minimum frequency that can still play the buffered content smoothly. The results for different video resolutions are shown in Table 2. Note that during our measurement, the background apps are still running as normal, and their CPU capacity requirement has also been considered.

TABLE 2
The minimum CPU frequency for video streaming when the wireless interface is turned off (Samsung Galaxy S5)

Video resolution	Min CPU frequency
360p	422 MHz
480p	652 MHz
720p	652 MHz
1080p	883 MHz

3.5 DASH-aware EFS Algorithm

Dynamic Adaptive Streaming over HTTP (DASH), also known as MPEG-DASH, is a widely adopted video streaming protocol to improve user QoE under unstable network condition [13]. A DASH client can switch between different bitrates adaptively based on the network condition, to ensure that the best-quality video can be downloaded in time for playback. Specifically, each task (video chunk) is encoded into multiple copies with a variety of discrete bitrates V, where a copy with higher bitrate indicates higher resolution (video quality) but a larger file to download. Suppose r_i is the network throughput measured when downloading T_i. The bitrate of T_{i+1} is selected as the highest bitrate less than r_i:

$$v = \arg\min_{\substack{v_j \in V \\ v_j \le r_i}} |v_j - r_i|. \tag{6}$$

Since the highest CPU frequency is used for data transmission by a traditional DASH client, r_i equals to the maximum network throughput r_{max}. Then the selected bitrate v will be the highest one allowed by the network condition.

To integrate our EFS algorithm with DASH, we need to consider the following problems. First, to save energy for task T_i, our EFS algorithm may reduce the CPU frequency for data transmission and thus the network throughput is also reduced. Then the measured network throughput r_i does not equal to the maximum network throughput r_{max}, so r_i cannot be used to determine the bitrate of T_{i+1}. To solve this problem, we use the TCP throughput model established in Section 2.3 to reversely estimate r_{max} based on r_i, i.e., $r_{max} = r_i / r^*(f_i)$, where f_i is the CPU frequency for data transmission of T_i. Then we can use r_{max} to determine the bitrate of T_{i+1}.

The second problem is to consider unstable network condition when downloading a task. If a task is downloaded using a CPU frequency pre-determined by prior network measurements, the task may not be downloaded in time when the network throughput drops, resulting in a rebuffing event [14]. To solve this problem, we propose an adaptive task downloader, which keeps monitoring the network throughput and adaptively adjusts the CPU frequency during a download. The pseudocode of the downloader is shown in Algorithm 1. For task T_i, the downloader first selects the bitrate v_i as the largest one allowed by the maximum network throughput, and determines the download-

Algorithm 1: Adaptive Task Downloader

```
Input: a task $T_i$ with video length $L_i$, maximum
       network throughput $r_{max}$ estimated when
       downloading the previous task $T_{i-1}$
$v_i$ = selectBitrate($r_{max}$)
Run EFS to determine the CPU frequency $f_i$ for $T_i$
Set frequency $f_i$
$L_{remain} = L_i$
$L_{chunk} = 1$ // 1 second video chunk
while $L_{remain} > 0$ do
    $d_{chunk} = v_i \times L_{chunk}$
    $t_{chunk}$ = downloadVideo($v_i$, $L_{remain}$, $L_{chunk}$)
    $r = d_{chunk}/t_{chunk}$
    if $r < v_i$ then
        $r_{max} = r/r^*(f_i)$
        $f_i$ = scaleUpFrequency($r_{max}$, $v_i$)
        Set frequency $f_i$
    end
    $L_{remain} -= L_{chunk}$
end

function selectBitrate($r_{max}$):
    $V$ = getAvailableBitrates()
    sort($V$) // sort in ascending order
    $m$ = size($V$)
    for $j = m$ to 1 do
        if $r_{max} > v_j$ then
            return $v_j$;
        end
    end
    return $v_1$;
end function

function scaleUpFrequency($r_{max}$, $v$):
    $F$ = getAvailableFrequencies()
    sort($F$) // sort in ascending order
    $m$ = size($F$)
    for $j = 1$ to $m$ do
        $r = r_{max} \times r^*(f_j)$
        if $r > v$ then
            return $f_j$;
        end
    end
    return $f_m$;
end function
```

ing CPU frequency f_i according to the EFS algorithm. It then starts downloading the video and estimates the network throughput every time a short video chunk (1 second) is downloaded. If the estimated network throughput is smaller than the bitrate v_i, the video may not be downloaded before playback begins. Then the downloader scales up the CPU frequency to increase the network throughput to guarantee that the video can be downloaded in time. The downloader also adaptively adjusts the bitrate if a significant change of the network condition is observed (not included in Algorithm 1 to simplify the presentation). To avoid short-term fluctuations, similar to [15], we use the weighted average network throughput, i.e., $r^w_{max}(j) = (1-\delta) \times r^w_{max}(j-1) + \delta \times r_{max}(j)$. where $r_{max}(j)$ is the measured throughput of interval j (i.e., downloading the jth video chunk), $r^w_{max}(j)$ is the weighted throughput measure of interval j, and δ is a weighting value, which is adaptively controlled based on the throughput deviation [16]. The bitrate is selected as the highest one allowed by $r^w_{max}(j)$. Other bitrate adaptation logics include buffer-based methods [4], [17], [18] and optimization-based approaches [19], [20], which can be integrated into our downloader.

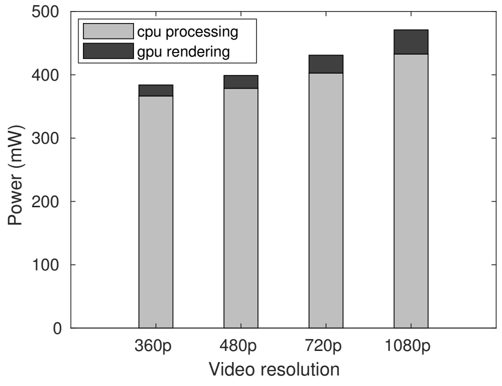


Fig. 8. The CPU power and the GPU power when playing videos with different resolutions on LG Nexus 5x

To adaptively adjust the video bitrate and the CPU frequency according to the network quality, We run the algorithms (i.e., EFS algorithm, bitrate adaptation algorithms) every time a significant change of the network condition is detected (i.e., the estimated max throughput $r^w_{max}(j)$ is 10% higher or lower than the prior one $r^w_{max}(j-1)$). Although running our algorithm once does not consume much energy (e.g., it takes 0.1 seconds to run the algorithm for a 30-minute video), when the network condition is unstable, frequently running the algorithms may lead to remarkable energy waste. Such energy waste is affected by the CPU frequency, and using a lower CPU frequency to run the algorithms does not always save energy. Fig. 7 shows the measured results of running our EFS algorithm at different frequencies on Samsung Galaxy S5 and Nexus 5X. As can be seen, using a lower CPU frequency consumes less power, but it takes more time to run the algorithms and may increase the energy consumption. The optimal CPU frequency that minimizes the energy of running the algorithms varies depending on devices, which is 1.19 GHz for Samsung Galaxy S5 and 0.96 GHz for Nexus 5x. Compared to the minimum frequency or the maximum frequency, using the optimal frequency can save energy by 30% to 50%. For each smartphone model, we need to find its optimal CPU frequency based on offline measurements and tune the CPU frequency when running the algorithms.

Rendering videos with different bitrates may lead to different GPU energy consumption. Fig. 8 compares the CPU power and the GPU power when playing videos with different resolutions on LG Nexus 5x. We first measured the whole phone power of playing videos with screen off. The CPU power was calculated using a utilization based model [21], in which the CPU power consumption is linear to the CPU utilization for a given frequency. Then the remaining

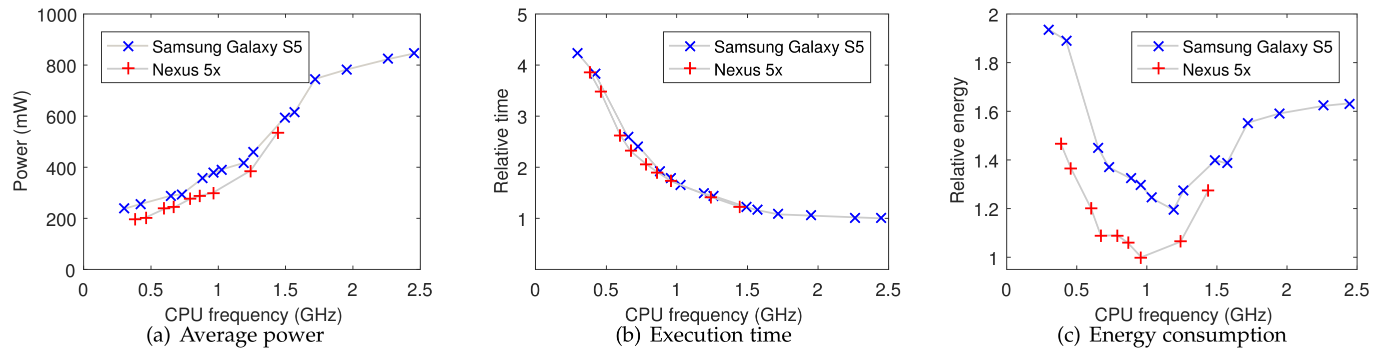


Fig. 7. Average power, execution time and total energy consumption of running the EFS algorithm at different frequencies on Samsung Galaxy S5 and Nexus 5X

power was the GPU power. As can be seem, the GPU power is much less than the CPU power. In this paper, we do not consider saving the GPU energy.

4 Energy-Aware Downloading Schedule for Video Streaming

The downloading schedule of video streaming determines when and how much data to download. However, the downloading schedule of existing apps is not optimized to reduce energy. For example, in Fig. 4(a), the downloading data size in most cases is small and therefore the wireless interface is frequently turned on, which consumes lots of energy. In this section, we design an energy efficient downloading schedule and combine it with our energy-aware CPU frequency scaling algorithm.

4.1 How Much to Download

Given a video size D, we can estimate its playback time based on the bitrate of the video. The bitrate v is related to the video resolution. For example, the 480p video has a bitrate of 1 Mbps and the 720p video has a bitrate of 2.5 Mbps [22]. The playback time for the video content is around D/v. The energy consumed by downloading D size of data may have four cases, as shown in Fig. 5. The minimum energy of using Case (a) and Case (b) to download different size of data is shown in Fig. 9. For both cases, the energy consumption is a straight line increasing with the data size. We call the data size when $E(T)^a = E(T)^b$ as β, and it is 1.5 MB in our case. As the energy of Case (c) and Case (d) is smaller than that of Case (a) and Case (b) by a constant value (the promotion energy), they are not considered here.

From Fig. 9, we can see that when the downloading data size is less than a threshold β, it should be downloaded by Case (b), i.e., it should be downloaded with a smaller throughput for a longer time. When the data size is larger than β, then it can be downloaded directly or divided into multiple pieces with each piece smaller than β. However, the energy per byte in Case (a) (the slope of the line) is much smaller than that of Case (b), and thus using Case (a) to download the same size of data is more energy efficient. Therefore, when the video content is larger than β, it should be downloaded in one piece using Case (a). This conclusion

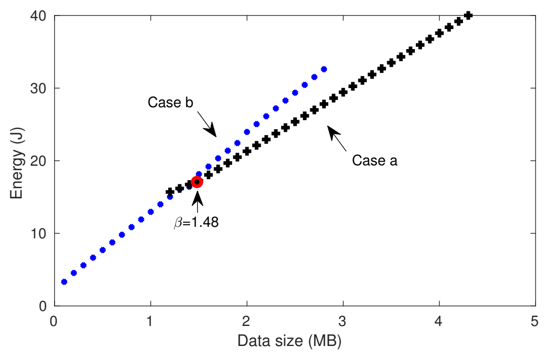


Fig. 9. The minimum energy to download video content with different data size

is also consistent with previous works [3], [4]. Considering the buffer to hold the video content on smartphones is limited, we set the optimal downloading data size to the maximum buffer size.

4.2 When to Download

As described in previous section, when the downloading data size D is larger than β, it is more energy efficient to download using Case (a), and then the LTE interface enters the idle state before the task end. To save energy, the LTE interface should stay in the idle state as long as possible. On the other hand, the video content should be downloaded before being used to provide better quality of experience (QoE). Thus, the next downloading should start a little earlier. The smallest decoding unit in video is called Group of Pictures (GoP), which has a fixed length according to the video coding protocol and frame organization [4]. Suppose this length is g, then the data size within the length of a GoP is approximately $v \times g$, where v is the bitrate of the video. As the CPU frequency used in the next downloading period is not known beforehand, we should consider the worst case, where the CPU frequency is f_{min} and the TCP throughput is $r(f_{min})$. To guarantee one GoP in the next task is downloaded before the end of the current task, the interval between tasks is computed as Eq. 7.

$$Interval = \frac{D}{v} - t_{pro} - \frac{v \times g}{r(f_{min})} \quad (7)$$

TABLE 3
Video benchmark

Video id	Length (sec)	Data size (MB)	Resolution
1	57	8.8	720p
2	163	20.3	480p
3	271	53.3	720p
4	301	39.7	480p
5	496	79.9	720p
6	594	56.1	480p

5 Evaluations

In this section we use trace-driven simulations to demonstrate that our energy-aware CPU frequency scaling algorithm can help existing video streaming apps to save energy, and more energy can be saved using the optimized downloading schedule. Also, we use power traces measured on real devices to verify the accuracy of our models.

5.1 Simulation Setup

The trace used for simulation is collected from the Youtube app running on Samsung Galaxy S5. We watch a group of videos with different length, data size and resolution, as listed in Table 3. We mainly consider videos less than 10 minutes since videos longer than 10 minutes are rare [9]. We collect two kinds of traces: the network trace, which is used to extract the downloading time and downloading data size, and the real-time CPU frequency trace, which is read from the file "`scaling_cur_freq`". Based on these traces, we mainly compare the performance of the following methods.

- `Youtube`: This method is the original Youtube app using the default interactive CPU governor to adjust the CPU frequency.
- `Youtube+MaxMin`: In this method, the Youtube app uses the highest CPU frequency during data transmission and the minimum CPU frequency as discussed in Section 3.4 when there is no data transmission.
- `Youtube+EFS`: This is the Youtube app which uses our *Energy-aware Frequency Scaling algorithm (EFS)* to adjust the CPU frequency, but the downloading schedule follows the default Youtube app.
- `Ourstreaming+EFS`: This method uses the optimized downloading schedule as described in Section 4, and also uses the EFS algorithm to adjust the CPU frequency. The buffer size is set to 10 MB.

5.2 Energy Comparison

The energy consumption is calculated based on the energy model established on Samsung Galaxy S5 and LG Nexus 5x (see Section 2.3). We first compare the whole phone's energy consumption of different methods when watching videos in Table 3, and show the results in Fig. 10. As can be seen, the energy consumption generally increases when the video length increases. This is because we consider the energy consumption during the whole playback period of the video. The MaxMin method saves a large amount of energy and the EFS method saves more. The combination of ourstreaming and EFS can save much more energy than simply using the EFS algorithm. When the video is longer and

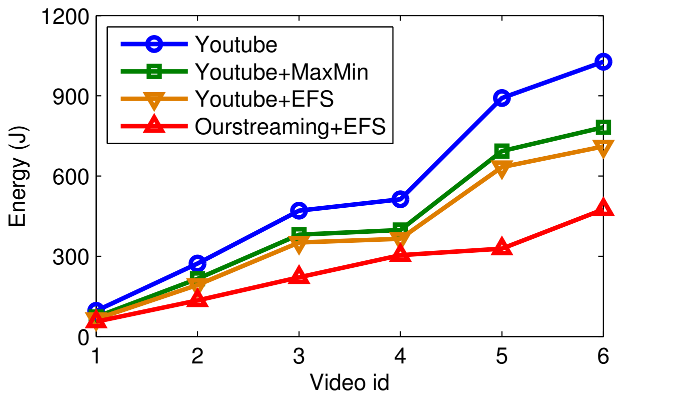


Fig. 10. The total energy consumption of different methods (calculated based on the energy model of Samsung Galaxy S5)

the data size is larger, more energy can be saved, because there are more downloading tasks and thus more opportunities to adjust the CPU frequency. On average, the MaxMin method and the EFS method can save 22.1% and 30.2% more energy than the default Youtube method, respectively. The combination of ourstreaming and EFS can save 50.6% of energy. On top of EFS, our optimized downloading schedule helps to save another 29.2% energy. We also notice that the energy consumption of MaxMin and EFS has similar trends as that of the default Youtube method, since they use the same downloading schedule. Ourstreaming adjusts the downloading schedule by transmitting multiple tasks together and thus shows a different trend. The energy of ourstreaming method is mainly related to the data size and the length of a video.

5.2.1 *Model Verification*

To verify the power and TCP throughput model (Samsung Galaxy S5 and LG Nexus 5x), we measure the energy consumption using the Monsoon power monitor. We watch a video at each CPU frequency and collect a set of measured power traces. For the MaxMin and EFS methods using the Youtube downloading schedule, we first compute the optimal CPU frequency in each downloading period using our method. Then for each downloading period from t_1 to t_2 using CPU frequency f, we extract the energy during t_1 and t_2 from the power trace when watching the video at CPU frequency f. By combining the energy consumption from multiple traces, we obtain the total energy. For ourstreaming+EFS, we download the video to a local server, and divide it into several parts with the data size specified by ourstreaming. We collect the energy consumption with data transmission by downloading each piece of the video at specified CPU frequency, and we collect the energy consumption without data transmission by watching the cached Youtube videos.

Here we show an example of video 2. The energy consumption of different methods based on the real power trace is shown in Table 4. Considering the default Youtube method, the results here are close to the simulation results. It indicates that our power model and TCP throughput model are pretty accurate, where the energy computed using our models only has an error rate of less than 8% for both

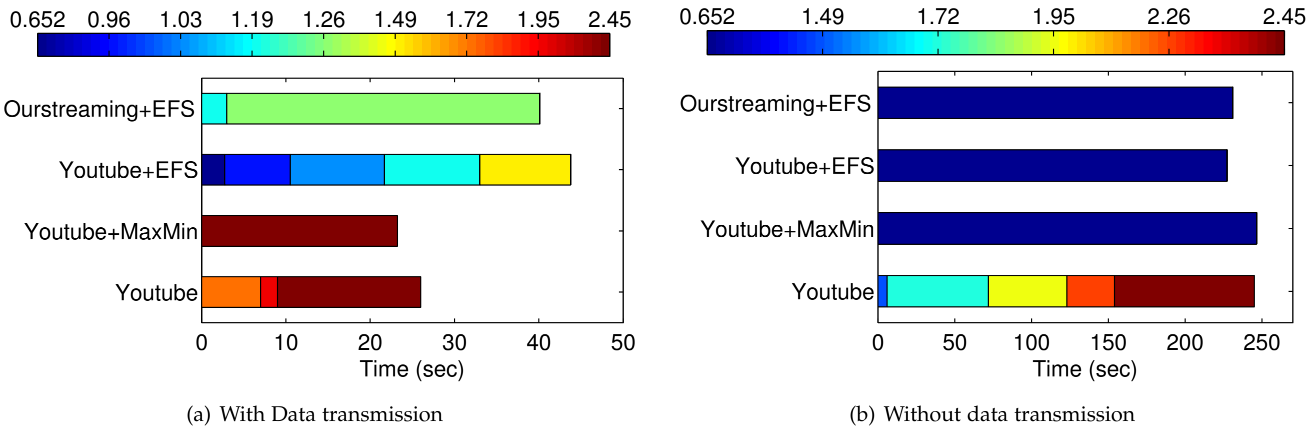


Fig. 11. The CPU frequency distribution using different methods with and w/o data transmission (Samsung Galaxy S5)

TABLE 4
The energy consumption of different methods based on real measurement

	Method	Energy (J)	Energy Saving Ratio
Galaxy S5	Youtube	295.1	-
	Youtube+MaxMin	230.3	21.9%
	Youtube + EFS	202.2	31.5%
	Ourstraming+EFS	146.2	50.5%
Nexus 5x	Youtube	278.4	-
	Youtube+MaxMin	209.4	24.8%
	Youtube + EFS	190.2	31.9%
	Ourstraming+EFS	152.2	45.3%

Samsung Galaxy S5 and LG Nexus 5x. From the results based on real measurement, we can see that compared to the default Youtube method, MaxMin saves 21.9% of energy on Samsung Galaxy S5 and 24.8% of energy on LG Nexus 5x; EFS saves 31.5% of energy and 31.9% of energy, respectively; Ourstreaming+EFS can save 50.5% of energy and 45.3% of energy. The results are also consistent with the simulation.

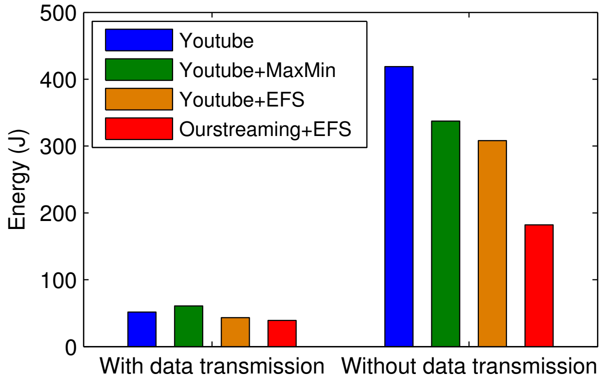
5.3 Impact of CPU Frequency on Energy

To better understand the energy saved by selecting different CPU frequency, we divide the total time into two time periods: the period with data transmission (promotion and data transmission time) and the period without data transmission (tail and idle time), and then analyze the CPU frequency distribution of each method in these two time periods. Take video 3 as an example, the CPU frequency distribution results are shown in Fig. 11. The length of a bar indicates the total time that the corresponding method spends in this time period. Each color inside the bar corresponds to one CPU frequency and the length of this color box indicates the time duration that the CPU works at the given frequency.

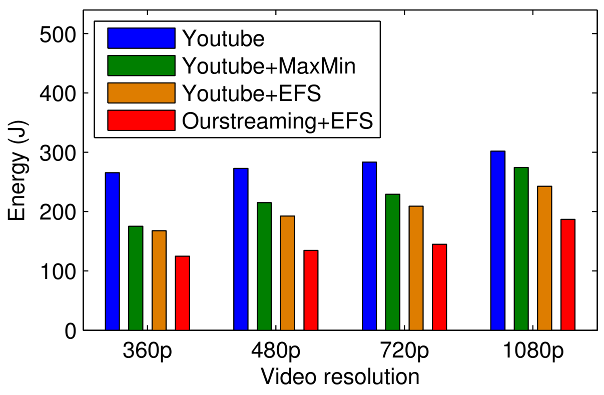
Figure 11(a) shows the CPU frequency distribution in the data transmission period. We can see the default CPU governor tends to set the CPU at a relative high frequency to get better performance, and the MaxMin method always sets the CPU to the highest frequency. High CPU frequency is helpful to reduce the data transmission time, but not the energy. In comparison, EFS searches for the best CPU frequency to minimize the data transmission energy and CPU energy, which is always much lower than the highest frequency. Though the data transmission time is longer, the power consumption is significantly reduced and then the total energy can be reduced.

As Youtube downloads different size of data and uses different downloading intervals, the optimal CPU frequency used by EFS during data transmission has large variation. For ourstreaming, it mainly downloads data to fill the whole buffer and uses a relative stable downloading interval, and thus the CPU frequency is similar. The last part of video content has a different size, so it uses a different CPU frequency. The CPU frequency distribution without data transmission is shown in Fig. 11(b). Both MaxMin and EFS use low CPU frequency. As a comparison, the default CPU governor adjusts the CPU frequency according to the CPU usage, and thus the CPU frequency has a large variation. Also, the default governor prefers to set the CPU frequency to relative high values. Although the default Youtube method has longer time without data transmission, its power during this period is too high and thus consumes more energy than EFS. From another point of view, when the CPU frequency is too high, many of its processing power is wasted.

Similar to the CPU frequency analysis, we also divide the total energy of each method into two parts: the energy with data transmission (including the promotion energy) and the energy without data transmission (tail energy and idle energy). The comparison of these two parts of energy of video 3 is shown in Fig. 12(a). As can be seen, the EFS algorithm can help save the data transmission energy, because it selects a proper CPU frequency. Specifically, Youtube+EFS saves 13.1% of energy and oustreaming+EFS saves 24.2% of energy during data transmission. However, MaxMin consumes a little more energy than the Youtube method since it always selects the highest (most power consuming) CPU frequency. When there is no data transmission, both MaxMin and EFS can save energy when compared to the Youtube method, because they all use lower CPU frequency. The EFS method reduces more energy than MaxMin when there is no data transmission because it reduces the time to

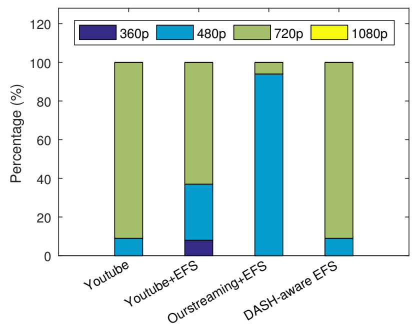


(a) With and without data transmission

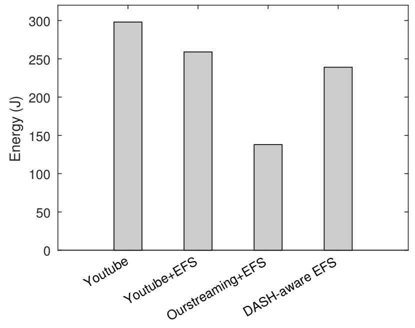


(b) Different resolutions

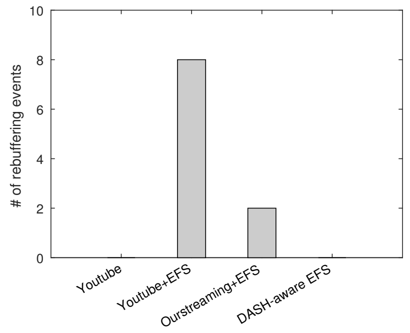
Fig. 12. The energy consumption of different methods during video streaming (Samsung Galaxy S5)



(a) Video resolution distribution



(b) Energy



(c) Rebuffering

Fig. 13. Video resolution distribution, energy and rebuffering during video streaming when DASH is enabled (Samsung Galaxy S5)

stay in this time period (Fig. 11(b)).

5.4 Impact of Video Resolution

Since mobile devices have different screen resolutions and different network speed, video providers generally provide multiple versions for the same video with different resolutions to satisfy users' requirements. The video client can select a fixed resolution or use DASH technology to dynamically adjust the video resolution. To test the performance of different methods under different video resolutions, we collect traces of video 2 with a resolution of 360p, 480p, 720p and 1080p, respectively. The energy consumption of different methods is shown in Fig. 12(b). Clearly, MaxMin and EFS can save energy under all resolutions, and ourstreaming+EFS can save more. Comparing the energy saving ratio of the same method among different resolutions, we can see that this ratio decreases when the video resolution increases. When watching the 360p version of video, MaxMin, EFS and ourstreaming+EFS can save 33%, 36.8% and 52.9% of energy when compared to the Youtube method, respectively. However, these saving ratios drop to 9%, 19.7% and 38.1% when watching the 1080p video. The reason is that a video with higher resolution has larger data size and more pixels, and thus all methods need to select a higher CPU frequency to download the video on time, decode and play smoothly. As a result, the difference between their CPU frequencies and the default system is smaller and less energy can be saved.

5.5 Evaluation of DASH-aware EFS Algorithm

DASH technology is used to adaptively adjust the video resolution according to the network condition. It is extremely useful to improve the video quality under an unstable network environment (e.g., cellular network). The DASH-aware EFS algorithm (discussed in Section 3.5) is designed to integrate our EFS algorithm with DASH in order to save energy while not sacrificing the video quality. We compare our DASH-aware EFS algorithm with other algorithms based on network traces collected under an unstable network environment (by downloading files from a mock server through LTE when walking around a building). Here we show an example of video 2 which has four available resolutions (bitrates): 360p (0.5 Mbps), 480p (1 Mbps), 720p (2.5 Mbps) and 1080p (5 Mbps). As shown in Fig. 13, Youtube displays 91% of the video at 720p resolution and 9% at 480p with no rebuffering event (rebuffering happens when the video content cannot be downloaded in time before the buffer runs out). EFS has much lower resolution (37% under 720p) than Youtube and it generates 8 rebuffering events, since EFS computes CPU frequency based on prior network measurements and thus it cannot react to network fluctuations timely. Ourstreaming+EFS tends to buffer low-quality video, leading to 94% of the video at 480p resolution. DASH-aware EFS displays the video at the same quality as Youtube does in terms of the video resolution distribution, since DASH is seamlessly integrated in the algorithm. By adjusting the CPU frequency, DASH-aware EFS can save 19.8% energy compared to Youtube.

6 Related Work

The wireless interface, especially the cellular interface consumes a lot of power on smartphones [23]. In cellular networks, the wireless interface stays in a high power state (tail state) after a data transmission, and the tail state wastes a large amount of energy. To save energy, researchers propose to aggregate data tasks together to amortize the tail energy [24]. Similar idea is also used by video streaming which downloads a group of video content together and then turn the wireless interface off [2]. However, video streaming has strict delay constraint and the data should be downloaded before being used [4], [25]. EVIS uses multiple networks to provide energy-efficient and quality-guaranteed video streaming [26], but it does not consider the impact of CPU frequency to the network throughput.

Video streaming requires lots of CPU processing power to provide good QoE. The CPU energy is related to its working frequency [5], [27]. High CPU frequency can provide better performance but consumes more energy. Many solutions have been proposed to adjust the CPU frequency to achieve a balance between performance and energy [28], [29], [30], [31]. They have some interesting results, such as how to select the CPU frequency to finish the tasks before their deadline and save energy [31], however, none of them considers the impact of CPU frequency on TCP throughput.

The energy consumed by video streaming includes data transmission energy and CPU energy. This makes the problem more complex, since the TCP throughput is closely related to CPU frequency [7]. High CPU frequency increases the CPU energy consumption, while low CPU frequency increases the data transmission time and may increase the data transmission energy. Kwak *et al.* consider the tradeoff between saving CPU energy and data transmission energy in [6], and suggest to reduce the CPU frequency when the network becomes bottleneck. Their intuition is to save the CPU energy when waiting for the network tasks. However, they do not consider that the TCP throughput is also reduced when reducing the CPU frequency. Thus, their solution may introduce too much delay for video streaming applications. In this paper, we consider the delay and set the CPU to a proper frequency that can save energy and ensure the video content is downloaded before being used.

7 Conclusion and Future Work

In this paper, we modeled the effects of CPU frequency on TCP throughput and system power, and studied how to save energy for video streaming considering the CPU frequency. During video streaming, high CPU frequency can reduce the data transmission time but it consumes more CPU energy; low CPU frequency reduces the CPU energy but increases the data transmission time and then increase the energy consumption. To address this problem, we proposed an energy-aware CPU frequency scaling algorithm (EFS) which can properly adjust the CPU frequency to reduce the overall energy during video streaming. This algorithm can be directly applied to existing video streaming apps, like Youtube. Also, the downloading schedule of existing apps is not optimized in terms of energy. We address this problem by proposing an energy efficient downloading schedule, which can save more energy when combined with the EFS algorithm. Based on trace-driven simulations and real measurement, we demonstrate that EFS can save 30% of energy than the default Youtube app. By using properly selected downloading data size and downloading interval in our EFS algorithm, more than 50% of energy can be saved when compared to the default Youtube app.

References

[1] "Cisco visual networking index: Global mobile data traffic forecast update, 2015-2020," http://goo.gl/DXWFyr, 2015.

[2] X. Li, M. Dong, Z. Ma, and F. C. Fernandes, "Greentube: power optimization for mobile videostreaming via dynamic cache management," in *Proc. 20th ACM Int. Conf. Multimedia*, 2012, pp. 279–288.

[3] M. A. Hoque, M. Siekkinen, and J. K. Nurminen, "Using crowd-sourced viewing statistics to save energy in wireless video streaming," in *Proc. ACM MobiCom*, 2013, pp. 377–388.

[4] W. Hu and G. Cao, "Energy-aware video streaming on smartphones," in *Proc. IEEE INFOCOM*, 2015, pp. 1185–1193.

[5] C.-H. Hsu, U. Kremer, and M. Hsiao, "Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors," in *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, 2001, pp. 275–278.

[6] J. Kwak, O. Choi, S. Chong, and P. Mohapatra, "Dynamic speed scaling for energy minimization in delay-tolerant smartphone applications," in *Proc. IEEE INFOCOM*, 2014, pp. 2292–2300.

[7] S. Sanadhya and R. Sivakumar, "Rethinking tcp flow control for smartphones and tablets," *Wireless Networks*, vol. 20, no. 7, pp. 2063–2080, Oct. 2014.

[8] "3c cpu manager," https://goo.gl/2OoLMF.

[9] X. Cheng, C. Dale, and J. Liu, "Statistics and social network of youtube videos," in *Proc. 16th Interntional Workshop on Quality of Service (IWQoS)*, 2008, pp. 229–238.

[10] S. Kumar, E. Hamed, D. Katabi, and L. Erran Li, "Lte radio analytics made easy and accessible," in *Proc. ACM SIGCOMM*, 2014, pp. 211–222.

[11] X. Xie, X. Zhang, S. Kumar, and L. E. Li, "pistream: Physical layer informed adaptive video streaming over lte," in *Proc. ACM MobiCom*, 2015, pp. 413–425.

[12] H. Nam, K. H. Kim, and H. Schulzrinne, "Qoe matters more than qos: Why people stop watching cat videos," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[13] S. Lederer, C. Müller, and C. Timmerer, "Dynamic adaptive streaming over http dataset," in *Proc. ACM MMSys*, 2012, pp. 89–94.

[14] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," in *Proc. ACM SIGCOMM*, 2014, pp. 187–198.

[15] T. C. Thang, H. T. Le, A. T. Pham, and Y. M. Ro, "An evaluation of bitrate adaptation methods for http live streaming," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 4, pp. 693–705, April 2014.

[16] T. C. Thang, Q. D. Ho, J. W. Kang, and A. T. Pham, "Adaptive streaming of audiovisual content using mpeg dash," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 1, pp. 78–85, February 2012.

[17] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," in *Proc. ACM SIGCOMM*, 2014, pp. 187–198.

[18] L. De Cicco and S. Mascolo, "An adaptive video streaming control system: Modeling, validation, and performance evaluation," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 526–539, Apr. 2014.

[19] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," in *Proc. ACM SIGCOMM*, 2015, pp. 325–338.

[20] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "Bola: Near-optimal bitrate adaptation for online videos," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[21] Y. Yang, Y. Geng, L. Qiu, W. Hu, and G. Cao, "Context-aware task offloading for wearable devices," in *IEEE ICCCN*, 2017.

[22] "Video bitrate," http://www.lighterra.com/papers/videoencodingh264.

[23] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," in *Proc. ACM Mobicom*, 2012, pp. 317–328.

[24] B. Zhao, W. Hu, Q. Zheng, and G. Cao, "Energy-aware web browsing on smartphones," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 761–774, March 2015.

[25] S. S. Krishnan and R. K. Sitaraman, "Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs," in *Proc. ACM IMC*, 2012, pp. 211–224.

[26] J. Wu, B. Cheng, and M. Wang, "Energy minimization for quality-constrained video with multipath tcp over heterogeneous wireless networks," in *Proc. IEEE ICDCS*, 2016, pp. 487–496.

[27] X. Ruan, X. Qin, Z. Zong, K. Bellam, and M. Nijim, "An energy-efficient scheduling algorithm using dynamic voltage scaling for parallel applications on clusters," in *Proc. IEEE ICCCN*, 2007, pp. 735–740.

[28] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proc. the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 89–102.

[29] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 18–28, Jan 2005.

[30] J. Zhuo and C. Chakrabarti, "Energy-efficient dynamic task scheduling algorithms for dvs systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, pp. 17:1–17:25, Jan. 2008.

[31] K. Kwon, S. Chae, and K. G. Woo, "An application-level energy-efficient scheduling for dynamic voltage and frequency scaling," in *Proc. IEEE International Conference on Consumer Electronics (ICCE)*, 2013, pp. 3–6.

Yi Yang received the BS and MS degrees in Computer Science and Technology from The Beijing University of Posts and Telecommunications in 2009 and 2012, respectively, and the PhD degree in computer science and engineering from the Pennsylvania State University, in 2018. His research interests include wireless networking, mobile energy optimization, mobile and cloud computing. He is a student member of the IEEE.

Wenjie Hu received the BS degree in computer science from Tongji University, in 2007, the MS degree in computer science from Tsinghua University, in 2010, and the PhD degree in computer science and engineering from the Pennsylvania State University, in 2016. His research interests include energy management for smartphones, mobile cloud, and mobile video. He is a student member of the IEEE.

Xianda Chen received the BS degree in software engineering from Northwestern Polytechnical University, China, in 2013, and the MS degree in electrical and computer engineering from Sungkyunkwan University, South Korea, in 2015, respectively. He is working towards the PhD degree in computer science and engineering at the Pennsylvania State University. His research interests include wireless networking, mobile energy optimization, and mobile video. He is a student member of the IEEE.

Guohong Cao is a Distinguished Professor in the Department of Computer Science and Engineering at the Pennsylvania State University. His research interests include wireless networks, mobile systems, wireless security and privacy, and Internet of Things. He has published more than 200 papers which have been cited over 19000 times, with an h-index of 74. He has served on the editorial board of IEEE Transactions on Mobile Computing, IEEE Transactions on Wireless Communications, and IEEE Transactions on Vehicular Technology, and has served on the organizing and technical program committees of many conferences, including the TPC Chair/Co-Chair of IEEE SRDS, MASS, and INFOCOM. He has received several best paper awards, the IEEE INFOCOM Test of Time award, and the NSF CAREER award. He is a Fellow of the IEEE.