Energy Efficient WiFi Display

Chi Zhang and Xinyu Zhang
University of Wisconsin-Madison

Ranveer Chandra
Microsoft Research

ABSTRACT

WiFi Display, also called Miracast, is an emerging technology that allows a mobile device (source) to duplicate its screen content to an external display (sink) via a peer-to-peer WiFi link. Despite its diverse application scenarios and growing popularity, Miracast consumes substantial power due to a combination of video encoding/decoding and transmission. In this paper, we first conduct a measurement study to quantify and model key parameters that scale Miracast's power consumption. We then propose a set of optimization mechanisms to bypass redundant codec operations, reduce video tail traffic, and relocate the Miracast channel dynamically to maximize transmission efficiency. We have implemented this energy-efficient Miracast framework on an Android smartphone. Experimental results show that the legacy Miracast system costs 1.3 to 2.4 Watts. Our framework reduces the power consumption by 29% to 61%, depending on the Miracast application's video traffic patterns. Our optimization mechanisms do not affect the video quality, and can even reduce the latency of certain Miracast applications.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

Keywords

WiFi Direct; WiFi Display; Miracast; energy efficiency; mobile phones

1. INTRODUCTION

Contemporary mobile devices feature a one-to-one binding with small display units. Amid the insufficient display real-estate, many application scenarios (*e.g.*, gallery sharing, collaborative editing, presentation) call for a dynamic binding towards extraneous, large displays, such as PC monitors, TV screens and wall-projectors. Urged by such demands for screencasting, the Miracast standard (also referred to as *WiFi Display*) was recently developed by the WiFi Alliance. Acting like a wireless HDMI cable, Miracast allows a user to, for example, echo display plus audio from a smartphone onto an external screen in real-time. Unlike proprietary video-cast solutions such as Apple Airplay or Google Chromecast, once Miracast is enabled, every UI component – from the general interface to videos – will be duplicated on the screen. Thus, no modification is required for applications.

MobiSys'15, May 18–22, 2015, Florence, Italy.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3494-5/15/05 ...$15.00.
http://dx.doi.org/10.1145/2742647.2742654.

WiFi-Direct, also referred to as WiFi P2P, serves as an enabling networking technology for Miracast. The Miracast source can mirror its local content via a peer-to-peer link with the sink, *e.g.*, a WiFi-enabled HDMI dongle connected to a TV screen. Meanwhile, it allows *tethered streaming*, *i.e.*, downloading content from an access point (AP) while streaming it to the sink. Such features offer easy compatibility with existing WiFi devices. In fact, latest mobile operating systems like Android [1] and Windows Phone [2] already have built-in support for Miracast. Corresponding sink adapters from Microsoft [3], Amazon [4], *etc.*, have recently witnessed a large consumer market.

However, for a battery-operated Miracast source like a smartphone, maintaining a wireless external display can be very costly. The source has to stream video frames through WiFi with high bit-rate towards the external display. Prior to sending, it has to encode (compress) the video stream so that the WiFi link is capable of handling the bit-rate. During tethered streaming, it also has to decode the downloaded content before compression and transmission. These activities together make Miracast power-hungry even when the mobile device is idle. The Miracast standard devised an optional energy-saving feature called *video frame skip*, which allows the source to stop generating/transmitting video frames if its screen content is static and not updating. However, this mechanism alone barely helps taming the energy cost, especially during continuous video streaming.

From a high level perspective, Miracast's power consumption involves many parts of the mobile device, including processing and networking, none of which can be trivially ignored. Despite a rich literature on mobile energy efficiency, Miracast necessitates new mechanisms to jointly optimize two aspects. First, existing work on mobile multimedia delivery (see [5] and the references therein) focused on curtailing the *receiver- or client-side* power consumption by reshaping the video server's traffic, thereby creating sleeping opportunities that can be harnessed by WiFi's Power Saving Mode (PSM). In contrast, Miracast's dominating operations reside on the source or *transmitter side*. Second, Miracast incurs substantial codec power consumption, especially in tethered streaming that requires real-time decoding and reencoding. It is the intersection between video-dominated application content generation and transmitter-centric network optimization that calls for a new, holistic approach towards energy-efficient Miracast.

This paper marks a first step towards systematic understanding and optimization of Miracast power consumption. We conduct a measurement of Miracast-compatible smartphones to obtain application-specific and component-wise power profile. The measurement results pinpoint key parameters such as screen resolution, video bit-rate, channel selection, which help establishing a power consumption model, later used in devising power-reduction strategies.

Grounded on the measurement insights, we explore a set of system optimizations to tame processing and networking related energy cost at the *Miracast source node*. First, for bursty screencast applications like slide show, we identify a video tail phenomenon that reflects a tradeoff between video quality and the source's traffic load that in turn affects its power consumption. We design an adaptive tail-cutting algorithm that reduces the power cost without noticeably impairing video quality.

Second, we explore a video pass-through mechanism that allows the source to circumvent decoding/reencoding of video frames, thus evading the power-hungry codec operations. For both encoded local content and network content downloaded via the AP (in tethered streaming mode), the source directly passes the video frames, and offloads the decoding tasks to the sink. This mechanism also enables batching and speculative transmission of video frames to the sink, without incurring the latency that used to appear when waiting for the codec to output frames at a slow, constant pace.

Third, observing that network contention affects the idle listening time and hence power efficiency of the source, we propose an off-channel Miracast scheme that opportunistically migrates the source-sink link to an energy-efficient channel. This so-called *off-channel* may differ from the AP-source channel in tethered streaming mode. Selection of the most energy-efficient channel is guided by the model that we have developed through measurement and calibration.

Besides, we have identified and curtailed background Miracast traffic due to silent audio generation and hidden image layers, which account for substantial energy wastage in certain applications with bursty video traffic patterns.

We have implemented the above power optimization mechanisms on Galaxy Nexus, a Miracast-compatible Android phone. The implementation is application transparent. Our experimental evaluation uses the Vanilla Android Miracast framework as a benchmark. The results demonstrate that the total system power savings range from 29% to 61% depending on application use cases and Miracast operation mode. Adaptive tail-cutting alone saves 2.3% to 19.6% for bursty applications. Video pass-through, combined with batching/prefetching, can save 52% to 61% for continuous video streaming. Off-channel miracasting can further add up to 8% for tethered streaming case under intensive contending traffic.

The main contribution of this work lies in a framework to optimize the *overall system-level power consumption* of a mobile device serving as Miracast source. This is feasible because when it is running, Miracast tends to dominate a mobile device's activities and computation/networking resources. Our specific contributions can be summarized as follows:

- Characterize Miracast power consumption in a component-wise manner, and develop a model to profile the impact of computation/network factors.
- Explore a set of power-optimization principles, including video tail cutting, video-frame pass-through, and energy-aware off-channel selection/switching. These principles also make it possible to execute conventional system-design principles, such as batching and prefetching, without hurting latency performance.
- Implement and evaluate the optimization mechanisms on top of Android's Miracast framework, in an application-transparent manner. Promising performance gains justify the proposed mechanisms, and demonstrate their potential as general guidelines for an energy-efficient Miracast system.

The remainder of this paper exposits each of the above contributions. Following background information about Miracast (Section 2), Section 3 describes our measurement and modeling of Miracast power consumption. Section 4 presents the power-optimization mechanisms with implementation, followed by Section 5 where we evaluate the energy savings. We discuss certain unexplored directions in Section 6, related works in Section 7, and finally conclude the paper in Section 8.

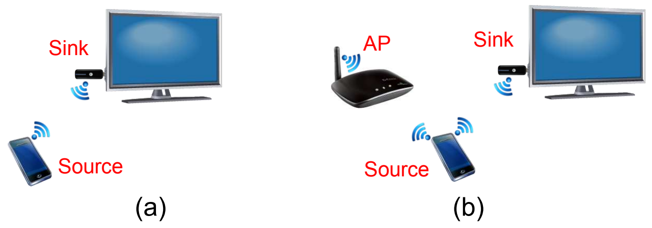

Figure 1: Network topologies for WiFi Display: (a) P2P topology for source's *local content streaming*; (b) Two-hop topology for *tethered streaming*.

2. BACKGROUND

2.1 WiFi Direct

WiFi Direct is a standard from the WiFi Alliance [6]. It enables devices to communicate by establishing P2P groups. Within the group, the device implementing AP functionality is called the *Group Owner (GO)*, whereas others are P2P *clients*. Group members negotiate roles during initial setup through a handshake with randomized ranking.

Two new power saving mechanisms are introduced for the GO. *Opportunistic power saving* mode allows the GO to sleep once all clients are asleep. Another mechanism, called the *Notice of Absence*, allows a GO to proactively announce time intervals where it powers down to save energy. In contrast to the GO, a WiFi Direct client is allowed to use legacy 802.11 Power Saving Mode (PSM).

2.2 Miracast: Wireless Display Over WiFi Direct

2.2.1 General Architecture

WiFi Display or Miracast [7] is another standard from the WiFi Alliance. It mirrors the screen of a device by streaming it as a live video over WiFi Direct to an external screen. Miracast enables a diverse set of use cases, such as: *(i)* Presentation slide or picture gallery show. *(ii)* Projecting GPS application in vehicles [8]. *(iii)* Mirroring videos to a TV screen or monitor. *(iv)* Multi-user gaming.

WiFi Direct's logical GO/client role assignment eases the deployment of two Miracast network topologies, as shown in Figure 1. In the *local streaming* topology, a mobile source node can mirror locally generated screen content, *e.g.*, general UI, game scenes, pictures and videos, towards the sink via a peer-to-peer connection. In the *tethered streaming* topology, the source downloads Internet content via the WiFi AP, renders it on the screen and casts it towards the sink. For compressed online video, the source needs to decode before rendering and casting. In both topologies, the source can act as either a GO or client *w.r.t.* the sink, but must be a client *w.r.t.* the infrastructure WiFi AP.

Miracast consists of multimedia protocols on top of the WiFi network stack as shown in Figure 2. The architecture is the same for the Miracast source and sink. The screen content is compressed into video frames and, along with audio frames, encapsulated in an MPEG2-TS container, and sent using UDP-based RTP. The TCP-based RTSP protocol provides session and playback control.

Figure 3 illustrates the source's workflow during a Miracast session on Android. Applications can generate screen content, which is composed with GPU and displayed on the UI. We emphasize that Miracast itself does not directly "capture" the screen. Instead, it leverages a media routing module to acquire the screen pixels directly, encodes them, and delivers them through the WiFi NIC. Whenever the screen rendering stops (*e.g.*, due to user turning off the screen), Miracast is also suspended.

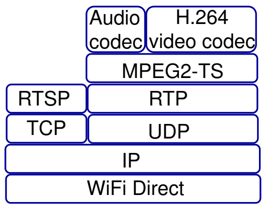

Figure 2: System architecture of Wi-Fi Display.

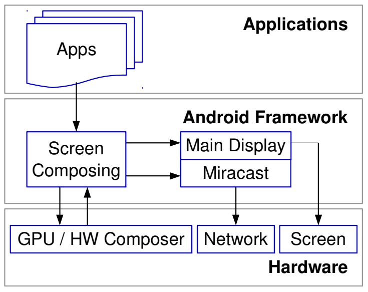

Figure 3: WiFi Display work flow on Android.

2.2.2 *Media Codec*

Screen mirroring can produce a huge traffic load. For example, the raw video stream of a screen with 1280x720 resolution, 16-bit color depth and 30fps refresh rate has a bit-rate of over 400Mbps, which is barely sustainable on current WiFi Direct. Thus the data must be compressed before casting over the P2P link.

Video encoding. Currently, H.264 is the only option for video encoding in the *Miracast* standard [7]. It is very efficient but also computationally intensive, and the CPUs on mobile devices usually lack the computational power to do the encoding in real-time. As a result, a dedicated hardware encoder is usually used, which can provide reasonable performance and relatively good energy-efficiency.

Audio encoding. Since audio data has a relatively low bitrate, compression of audio stream is optional [7]. There are 3 available formats: uncompressed LPCM, AAC and Dolby AC-3. Audio encoding needs to rely on CPU if the device does not have a corresponding hardware encoder. However, the CPU time consumption is usually only a small fraction of the total CPU time.

2.2.3 *Power Saving Mechanisms in Miracast*

Miracast proposes a new technique called Video Frame Skipping (VFS) [7] to save power. This is useful when the source's screen is temporarily static, in which case capturing the screen and casting it continuously will waste substantial channel time, and hence energy. VFS avoids such wastage by allowing the source to stop casting if it detects a static screen. The decision of how to detect and invoke VFS is left to the device vendor. Note that even when VFS is triggered, the RTSP still needs to send heart-beat messages (once per 25 seconds in Android 4.2) to keep the TCP control session alive.

3. UNDERSTANDING THE POWER CONSUMPTION OF WIFI DISPLAY

In this section, we present a measurement study that quantifies the energy overhead of Miracast. We then build simple models to pinpoint key variables that can be optimized.

3.1 A Breakdown of Power Consumption

Measurement setup. We use the Samsung Galaxy Nexus (GT-I9250) phone as our main testbed. It runs the Android Open Source Project (AOSP) 4.2.2 and can act as either a Miracast source or sink. Its hardware codec provides 30fps HD video encoding, and around 100fps of playback rate in H.264 format. In its default Miracast setting, the codec encodes videos with 720p resolution and a target video bit-rate of 5 Mbps. The Miracast implementation of AOSP 4.2.2 comes with the VFS mechanism and is enabled by default, and we use it as the baseline.

We use the Monsoon Power Monitor [9] to measure power consumption in real time when the Galaxy Nexus acts as a Miracast

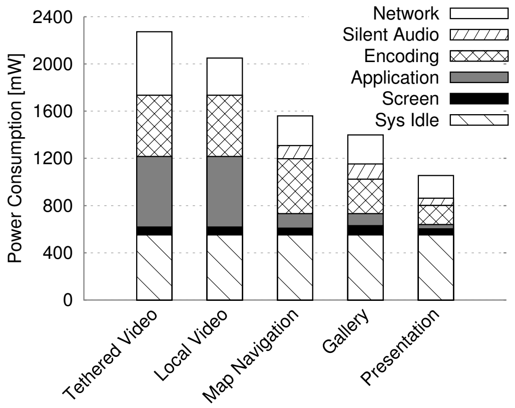

Figure 4: A breakdown of Miracast's power consumption. Application workload decreases from left to right.

source. For accuracy and consistency, we strive to eliminate irrelevant factors that may lead to random results. We always keep the phone in airplane mode to prevent the cellular modem from interfering with our measurements. We also turn off Bluetooth and GPS and screen auto-rotation unless they are required by the specific test. Screen brightness is set to the minimum by default. This mimics real scenarios – when a user is watching the mirrored screen, the phone tends to be left untouched, and thus can be switched to a low brightness state.

To obtain a component-wise breakdown of power consumption, ideally an instrumented phone is needed. Due to limited access to the Galaxy Nexus hardware, we approximate the breakdown by incrementally enabling system components. This enables measurement of 6 modules constituting Miracast: sys idle, screen, application, encoding, network, and silent audio.

We measure the *sys idle power* by blacking out the screen while keeping the phone awake in an idle state. To isolate irrelevant background applications, we install a clean AOSP on the phone without the Google services and also disable the dynamic wallpaper. To obtain *screen power*, we built an application that lights up all pixels from black state to the default brightness level. We measure the corresponding power increment. Galaxy Nexus has an AMOLED screen, whose power consumption depends on contents and may vary across different applications. To obtain the *application power*, we run the application but without Miracast, and then measure the power consumption increment to the sys idle and screen. To measure the *encoding power*, we put the network interface in sleep state, run the Miracast application, intercept and drop all video frames it generates to prevent them from triggering network operations. Again, the power increment is used to approximate the encoding power.

Besides, Miracast continuously delivers background audio even though the application is silent (details in Section 4.5.2). Correspondingly, it induces additional *silent audio power*. Finally, we isolate the *network power* consumption by enabling the source-sink miracasting and measuring the additional power on top of the prior 5 components.

Measured Miracast applications. We profile the following 5 typical Miracast applications.

(i) Tethered video: a popular video clip on YouTube is streamed to the source in real-time while being cast to the sink. *(ii) Local video*: a downloaded version of the YouTube video is used by the source for local playback and direct miracasting to the sink. *(iii) Gallery*: a random set of images from the Bing gallery, updated every 2 seconds. *(iv) Presentation*: 25 black-on-white slides, some with figures, updated every 15 seconds. *(v) Map navigation*: We use the OpenStreetMap data to design a route that fits on to a few

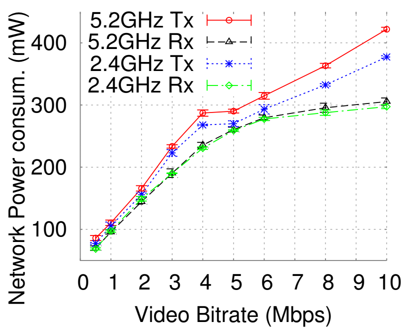

Figure 5: Measured dynamic power from Miracast network interface.

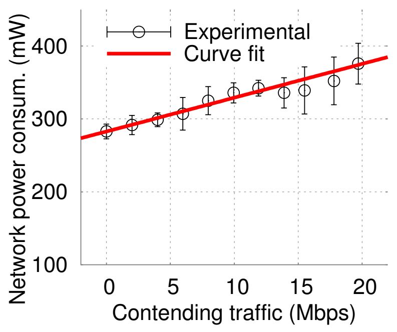

Figure 6: Measured network power consumption vs. contention intensity.

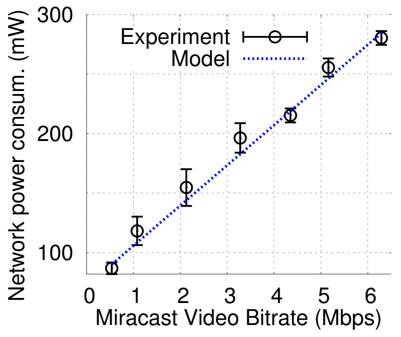

Figure 7: Measured network power consumption vs. Miracast video traffic load.

roads with at least 2 turns, covering middle to west part of the UW-Madison campus. We built an app that stitches, pans and renders the map tiles in real-time with 1Hz update rate, typical for GPS applications. Each tile has a resolution of 256×256 pixels and the route covers 100 tiles.

Breakdown of Miracast power consumption. Figure 4 presents the averaged power numbers. System idling consumes approximately 557 ± 10 mW, which accounts for a nontrivial fraction in the 3 applications with intermittent, bursty Miracast traffic (map, gallery and presentation). The screen power consumption falls in the range of 40 - 100mW, a relatively small fraction in all test scenes[1].

The application power consumption is negligible (35 mW or 3.2%) in the Presentation, but increases with the application's workload. Between the bursty application and continuous video application that involves a hardware codec, there is a drastic increase of application power (from below 120 mW to around 600 mW). This trend implies that the majority of the application power may be attributed to the codec's decoding operations, since other hardware components (*e.g.*, CPU and memory) are only handling lightweight tasks (*e.g.*, audio encoding and occasional user interaction).

The encoding power shows a similar increasing trend, accounting for 162 mW, 291 mW, 465 mW, 519 mW and 519 mW in the 5 applications with increasing work load. Accordingly, the Miracast video traffic becomes more intensive, resulting in increasing network power consumption (192 to 315 mW). Also note that the silent audio accounts for a non-negligible fraction in the 3 bursty video applications.

Overall, aside from the sys idle and screen power that is independent of Miracast, other modules together account for 450 mW to 1656 mW, or 43% to 73% depending on applications – thereby providing sufficient opportunities for power optimization.

3.2 Modeling WiFi Display Power Consumption

We aim to build a coarse model to highlight the key factors that dominate Miracast's asymptotic power consumption at the source. We focus on the local streaming mode. The tethered streaming differs only by adding a legacy (AP-source) WiFi link. Our model accounts for two major power consumers: network and codec.

3.2.1 *Network Factors*

Our network model estimates the energy overhead when performing Miracast on different channels (with different contention levels and hardware profile). This estimation will later facilitate our off-channel miracasting scheme. In contrast to prior analytical work that focused on 802.11 MAC layer [10, 11] or cognitive radios [12, 13], our model is measurement-driven and explicitly accounts for the impact of channel selection on power consumption.

Model. For a given channel and traffic load on the source-sink link, mean power consumption of the source can be dissected into 2 parts: P_{static}, the average *static power* or idle power to keep the network interface ready for transmission/receiving, and P_{dynamic}, the dynamic power, or *power increment* on top of P_{static} when the source is actively transmitting/receiving. The total network power is the sum of P_{static} and P_{dynamic}:

$$P_{\text{network}}(Ch, B_i) = P_{\text{static}}(Ch, B_i) + P_{\text{dynamic}}(Ch, B_i) \qquad (1)$$

where $B_i\,(i = t, r)$ denotes the traffic load in bps, for transmitting and receiving, respectively. For simplicity, we represent the total traffic load as $B = \sum_{i=t,r} B_i = B_t + B_r$.

The static part can be further decomposed as follows:

$$\begin{aligned} P_{\text{static}}(Ch, B) &= P_{\text{bs}}(Ch) \cdot R_{\text{active}}(Ch, B) \\ &= P_{\text{bs}}(Ch) \cdot \left(R_{\text{wait}}(Ch) + \frac{B}{B_{\text{PHY}} \cdot \eta_{\text{MAC}}} \right) \qquad (2) \end{aligned}$$

where R_{active} denotes duty-cycle, or the fraction of time the NIC is not sleeping. $P_{bs}(Ch)$ is the base power, which only depends on WiFi hardware and may vary as it is tuned to different channels. The duty-cycle consists of the fraction of idle time $R_{\text{wait}}(Ch)$ plus the fraction in transmission/receiving. The former mainly depends on contention intensity on a given channel Ch. The latter depends on traffic load B, WiFi bit-rate B_{PHY}, as well as MAC efficiency η_{MAC} (*i.e.*, ratio between MAC throughput and PHY bit-rate). In practice, since the source-sink distance tend to be short, and both tend to be relatively stationary, B_{PHY} usually remains stable. Thus, the fraction of time in transmission/reception only depends on traffic load B.

For the dynamic component, suppose $E_i(Ch)$ is the additional energy per bit needed on top of static power for channel Ch, then,

$$P_{\text{dynamic}}(Ch, B_i) = \sum_{i=t,r} [E_i(Ch) \times B_i] \qquad (3)$$

This component also accounts for related CPU power budget. $E_i(Ch)$ $(i = t, r)$ can be obtained by running a constant-bit-rate UDP session with PSM disabled, and subtracting measured $P_{\text{bs}}(\text{Ch})$ from aggregate power.

Combining Eq. (2) and (3), we can pinpoint 4 factors to determine the network power consumption: B, $P_{\text{bs}}(Ch)$, $R_{\text{wait}}(Ch)$ and $E_i(Ch)$. For the 3 channel-dependent factors, only $R_{\text{wait}}(Ch)$ depends on contention intensity in the radio environment. $P_{\text{bs}}(Ch)$ and $E_i(Ch)$ can be measured and tabulated for each device model during factory calibration. Note that this model ignores the sleeping power. In

[1]However, at full brightness with all pixels being white, the AMOLED screen can consume 765.9 ± 1.3mW.

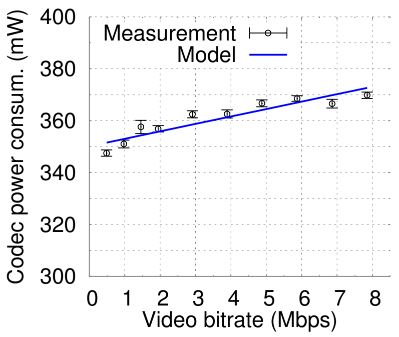

Figure 8: Measured decoding power vs. video bit-rate.

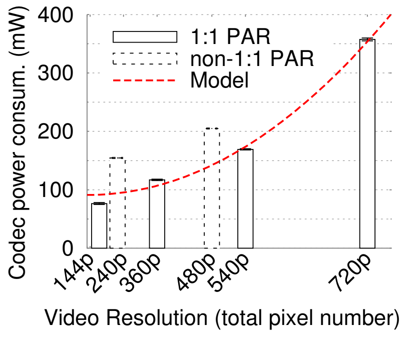

Figure 9: Decoding power vs. video resolution. Video bit-rate is 1.5 Mbps.

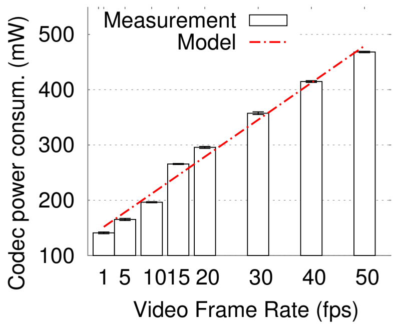

Figure 10: Measured decoding power vs. frame-rate.

addition, it does not explicitly model PSM, but the impact of PSM is to reduce idle listening power, which is indirectly reflected in $R_{\text{wait}}(Ch)$.

Validation and observations. We validate the above modeling approach on Galaxy Nexus. We first calibrate the $P_{\text{bs}}(Ch)$ and $E_i(Ch)$ parameters for this hardware model. $P_{\text{bs}}(Ch)$ is obtained by measuring the power difference between the case when the NIC is sleeping and when it is idle. Idle mode is created by disabling PSM while no packet transmission is going on. To obtain $E_i(Ch)$, we set up a WiFi Direct link between two phones on channel Ch, and run the *iperf* utility to generate a specific traffic load B. We then obtain $P_{\text{dynamic}}(Ch,B_i)$ by measuring the power increment on top of the $P_{\text{bs}}(Ch)$. $E_i(Ch)$ is taken as the first-order derivative of P_{dynamic} with respect to B_i. Our experiments are done in late night to minimize the impact from ambient traffic.

The *dynamic power model* assumes a linear relation between $E_i(Ch)$ and P_{dynamic}, which we verify through measurement (Figure 5). We observe that $P_{\text{bs}}(Ch,B_t)$ is roughly linear *w.r.t.* B_t, except when B_t falls between 4 and 5 Mbps, likely because of boundary effects caused by frame fragmentation. $P_{\text{bs}}(Ch,B_r)$ is close to linear when $B_r < 5$ Mbps, but levels off afterwards, likely because of interrupt aggregation for the network I/O. However, for a Miracast source, transmission traffic dominates and B_r typically falls well below 5 Mbps. From Figure 5, we also note that 5.2GHz band consumes more dynamic power than 2.4GHz, which is attributed to radio hardware. We also observe that different channels within the 5 GHz band or 2.4 GHz band show similar dynamic power (curve omitted here). Thus, $E_i(Ch)$ only needs to be calibrated for one channel on each band.

To verify the *static power model*, we measure the network power under different contention intensity and hence R_{wait} values, generated by varying the traffic load of an external link on the same channel. The traffic of Miracast is held constant at 5Mbps, such that the dynamic power remains constant and the power increase only comes from the static part. From the measurement results (Figure 6), we see contention intensity and P_{static} fit a linear relation, consistent with the model in Eq. (2). In Section 4.4.1, we will develop a channel surveying mechanism that enables a Miracast source to estimate R_{wait} directly.

Finally, Figure 7 depicts the Miracast source's *total network power* consumption P_{network} as video transmission traffic B_t increases (the source's receiving traffic B_r is negligible). Due to linear relation between B_t and static/dynamic power, the total power fits a linear relation as well.

Since $P_{\text{dynamic}}(Ch,B_t)$ is already measured, we can subtract it from $P_{\text{network}}(Ch,B_t)$ to obtain $P_{\text{static}}(Ch,B_t)$. Based on Eq. (2), we know the slope of the measured curve in Figure 7 equals $P_{bs}(Ch)/(B_{\text{PHY}}\eta_{\text{MAC}})$. Since $P_{bs}(Ch)$ and B_{PHY} are known, we can also obtain η_{MAC}. By now, all stationary parameters have been calibrated and the model (1) can be used to compare the relative power consumption when the Miracast source runs a different Ch and video bit-rate B_t.

3.2.2 Codec Factors

Model. We model the codec power consumption as:

$$P_{\text{codec}}(R_i,B_i) = \sum_{i=e,d} \left(E_{fi}(W\times H)\cdot R_i + E_{bi}\cdot B_i\right) \qquad (4)$$

where R_i, B_i and $W \times H$ represent the video frame-rate, bit-rate, and resolution, respectively. When VFS is enabled, we take the time-averaged values for R_i and B_i. Also, $E_{fi} \quad (i=e,d)$ is the energy per frame that the codec consumes for encoding and decoding, respectively. $E_{bi} \quad (i=e,d)$ is the overhead energy per bit, mainly associated with memory movements and extra CPU computations. Note that audio codec is omitted by this model considering the relatively small power consumption.

Validation and observations. By playing a set of video clips encoded with different $W\times H$, R_i and B_i from the same lossless video source, we can measure the decoding power and thereby calibrating E_{fi} and E_{bi} $(i=d)$. The measurement methodology follows Section 3.1.

Figure 8 shows that the decoding power consumption indeed follows a *linear relation* with B_i as modeled in Eq. 4, but the slope E_{bi} is less than 3mW/Mbps. For a typical 720p video stream, corresponding bit-rate ranges from 2 to 8 Mbps. Within this range of bit-rate, E_{bi} only contributes to a small fraction of the energy cost.

In contrast, the video resolution $W\times H$ affects the power consumption much, as shown in Figure 9. This is because the amount of computation scales non-linearly with number of pixels. In effect, the E_{fi} fits a *square function* with resolution. Notably, the additional processing power is needed for videos with a pixel aspect ratio (PAR) other than 1:1. For example, 480p video with a 16:9 display aspect ratio can not have a square pixel shape since $(480/9)\times 16$ is not an integer. Thus, it consumes even more power than 540p.

Consistent with our model, measurement results in Figure 10 verify that decoding power grows *linearly* with frame rate, since the codec needs to sustain more workload per unit time. The base power component caused by the playback (CPU, GPU, screen, *etc.*) is estimated to be about 145.1mW.

We have observed similar trend in encoding power *w.r.t.* B_i, R_i and $W\times H$, albeit with different model parameters. We omit the measurement details due to space constraint. Notably, the Miracast encoder on Android defaults to a fixed resolution of 720p, target bit-rate 5 Mbps and frame rate 30fps. Time-averaged B_i and R_i differ across applications and corresponding encoding power has been discussed in Section 3.1.

4. ENERGY EFFICIENT MIRACAST SYSTEM

4.1 Overview of Solution Set

We follow a set of simple principles to optimize the Miracast source's power consumption: when idle, maximize sleeping time; when busy, work in the most efficient way and cut unnecessary transmissions; amortize the cost of other states and offload tasks to the energy-insensitive sink node. Table 1 summarizes our specific designs and the applications, categorized according to network topology as local/tethered streaming, and according to traffic pattern as continuous/bursty video.

Our first solution mechanism, adaptive video tail cutting, reduces redundant transmissions that follow each traffic burst in intermittent screen-update applications (*e.g.*, map, gallery or slide show). Such redundancy is used to refine the quality of video frames, but at the cost of substantial traffic load and hence power consumption.

The second mechanism, video pass-through, applies to continuous video traffic. It allows the source to bypass the H.264 encoding and offload the decoding task as well to the sink node. Rather than waiting for the codec, the source can also batch a group of already coded frames and send them to the sink in a speculative manner.

In addition, we explore an off-channel Miracast mechanism, that executes an energy-aware channel selection scheme for the source-sink WiFi Direct link, based on the model in Section 3.2. In tethered streaming scenario, the source can alternate between the channel with AP (which may not be controllable), and the energy-efficient channel it selects. With this measure, it also avoids self-contention between the AP-source link and source-sink link, allowing for more efficient video data transportation.

In our system, the Miracast source acts as a client *w.r.t.* the sink and/or the AP. Thus, all the above mechanisms need to work coherently with the PSM protocol. Yet PSM itself is unaware of Miracast traffic patterns. We design a *PSMlock* mechanism, to manage PSM in a fine-grained manner following the typical clustered patterns of Miracast video traffic, thus creating more sleeping opportunities. Besides, background audio/video frames, invisible but presented to the Miracast source due to oblivious screen rendering driver, can be suppressed to save substantial power.

4.2 Cutting Video Tails

4.2.1 *Video Tail: Quality and Power Tradeoff*

The VFS allows a Miracast source to pause encoding and casting video frames when there are no updates on the screen. However, it is left to the vendor to decide when to pause. A trivial way is to encode one video frame for a still screen, and stop immediately afterwards. However, this may significantly impair video quality. For a sophisticated scene, the H.264 codec is unable to compress all the details into a single frame. Thus, the "encode-and-stop" approach may result in an incomplete or low-quality image delivered to the sink.

To deal with the problem, the Miracast source needs to add a "tail" after each sudden scene transition to gracefully refine the image quality. It keeps the encoder running and generating new frames until there are enough details in the image for human observation. In Android's Miracast framework, the tail is fixed to 30 frames per screen, lasting 1 second after each screen transition.

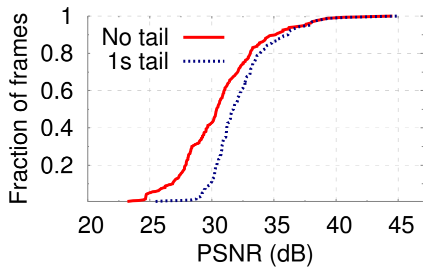

Figure 11: PSNR distribution with and without inserting video tail.

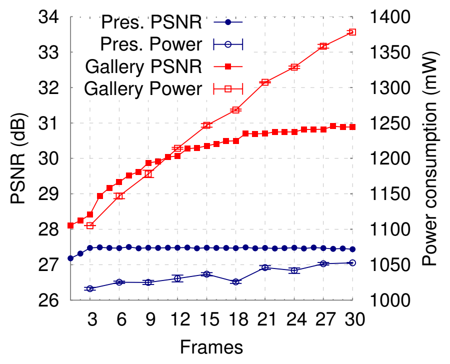

Figure 12: Additional power consumption due to video tail, and the resulting PSNR increase.

To gain a quantitative understanding of impact of tail on video quality, we rerun the gallery application with 0.5 Hz picture transition rate, using another Galaxy Nexus as sink node. We dump the sink's screen image data when each screen is encoded by one frame ("no tail" case) or 30 frames ("1s tail" case). Then we compute the PSNR by comparing with the original image at source side. Figure 11 plots the CDF of PSNR across all pictures. We see a median PSNR improvement of 1.1 dB and best case 5 dB, which upgrades the corresponding perceived picture quality at the sink.

On the other hand, a video tail generates additional frames that translate into energy cost. Figure 12 plots the average power consumption and PSNR as a function of tail length. For the gallery application, whereas PSNR increases by up to 3 dB, power consumption increases quickly – by 280 mW as tail length grows from 1 to 30 frames. For the presentation slide show, the power consumption increase is less obvious due to long idle period between 2 slides, within which network management overhead (*e.g.*, beacons and idle waiting) dominates the additional tail energy.

Note that for fast-changing scenes, like a 30fps continuous video casting, tails are not needed, as human eyes can not capture fine details before each next scene is displayed. However, for bursty video casting like slide show, we can identify much more details since each scene will be displayed for an extended amount of time to become discernible.

4.2.2 *Adaptive Tail Cutting*

Given the tradeoff between video quality and power consumption for certain bursty miracasting traffic, an adaptive algorithm is preferred that cuts the video tail but without hampering PSNR. A straightforward solution is to compute PSNR on-the-fly for each tail frame output by the codec, and stop encoding once PSNR plateaus. However, the computational cost is formidable. Alternatively, we can monitor the codec's output frame size to gauge how much information is included in the frame and whether more details need to be added to the picture. However, the relationship of encoded

	Adaptive video tail cutting	Video pass-through	batching & prefetching	off-channel miracast	PSMlock	background suppression
Local streaming	✓	✓	✓	✓	✓	✓
Tethered streaming	✓	✓	-	✓	✓	✓
Continuous video	-	✓	✓	✓	✓	✓
Bursty video	✓	-	-	✓	✓	✓

Table 1: Solution space for energy efficient Miracast.

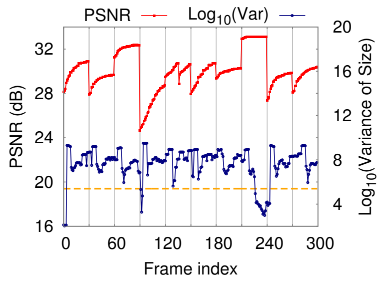

Figure 13: PSNR and variance of tail frame size, across 10 example video scenes with 30 tail frames each, lasting 1 sec. The dash line indicates the threshold for cutting tail.

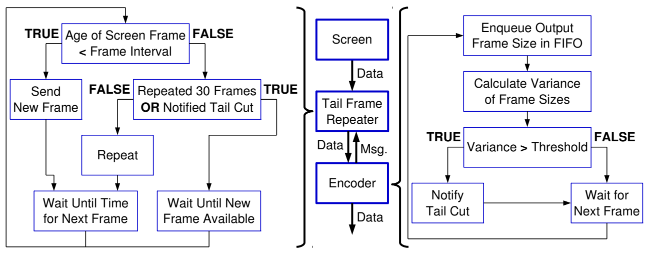
TRUE
Age of Screen Frame < Frame Interval
FALSE
Send New Frame
FALSE
Repeated 30 Frames OR Notified Tail Cut
TRUE
Repeat
Wait Until Time for Next Frame
Wait Until New Frame Available
Screen
Data
Tail Frame Repeater
Data
Msg.
Encoder
Data
Enqueue Output Frame Size in FIFO
Calculate Variance of Frame Sizes
TRUE
Variance > Threshold
FALSE
Notify Tail Cut
Wait for Next Frame

Figure 14: Work flow of the adaptive tail cutting implementation.

frame data size with the room of PSNR improvement is not straightforward – we found that absolute frame size is not monotonically decreasing as more tail frames are added.

Fortunately, we empirically observed that the *variance* of the output frame sizes will drop when PSNR stops growing. Figure 13 plots the PSNR growth with the tail length, in contrast to how variance of frame sizes changes over time, which testifies our observation. We suspect the phenomenon is caused by the interaction between the codec's bit-rate control and quantization. The H.264 encoder works to meet a fixed target video bit-rate constraint in Miracast (default to 5 Mbps in Android 4.2). Under this constraint, immediately after a scene change, the encoder oscillates between low/high quantization level and large/small frame size. The oscillation stops after PSNR saturates, because there is insufficient difference to encode to generate a large frame, and thus the frame size remains roughly consistent.

Our adaptive tail cutting algorithm uses the variance of tail frame size as a decision metric. We keep a moving window to calculate the size variance of 5 most recent frames, and cut the tail once the variance drops below a threshold. We found an empirical threshold of 250000 (corresponding to standard deviation 500 bits) works across a wide range of tested applications, although online tuning may yield even better quality/power tradeoff. Also, we upper-bound the tail length to 30 frames, corresponding to the 1 second tail in Android's Miracast framework. Figure 14 illustrates the integration between our algorithm and modules inside Android's Miracast framework, where a repeater keeps feeding the encoder with frames until tail ends. Decision of tail cutting in our algorithm is made by the encoder which notifies the repeater via a message interface.

4.3 Video Pass-Through

In this section, we describe the video pass-through mechanism that opportunistically offloads the source's codec tasks to the sink, while allowing batching and speculative transmission of videos as raw data.

4.3.1 Workflow and Implementation

Our pass-through design is application-transparent: it detects H.264 coded video data from application, intercepts the frames, and passes them directly to the Miracast module without involving the codec. We note that this scheme precludes the casting of screen elements other than the video.

To understand how video pass-through works, we first introduce the source's default way of mirroring local/online video (Figure 15(a)). An application typically calls a multimedia player framework to read coded video data from either local file or network I/O. The player further leverages a codec interface to invoke the hardware codec which *decodes* the video data. The video frames are rendered on the source's screen and mixed with other UI elements, if any, through a composer module. The composite frames are passed to the Miracast module as screen content. Then, Miracast follows its standard procedure to *encode* the content using the hardware codec, and eventually deliver the coded data to the sink.

In the pass-through design, we add a format checker on the codec interface, and pause the hardware codec whenever an H.264 video is played (Figure 15(b)). Then, the video data are packetized directly and fast-tracked to the network interface, as if they are the codec output under the command of Miracast. However, before that, we need two manipulations over the video frames on behalf of the legacy Miracast module, so as to hide the pass-through operation from the sink, allowing it to play video as usual.

First, the passed-through video frames have their native timestamps relative to the video start point. To cast the video in a Miracast session, we need to replace the native timestamps with the Miracast timestamp, such that the video can be played in the correct order together with other screen-generated frames before/after the video.

Second, the passed-through frames also have their native sequence numbers. Ideally, these should be modified as well to match the Miracast video sequence. However, we found the sequence number can have variable length, from 4 bits used in Internet videos to 16 bits in Galaxy Nexus. Worse still, the offset of the sequence number within each video frame is unknown, because H.264 uses Golomb coding to encode the parameters field prior to the sequence number in a video frame's header.

We overcome this hurdle by forcing the source to execute certain reset operations. Specifically, when pass-through starts, the source composes an *end-of-sequence frame* to tell the sink to reset the sequence number, and a *parameter frame* to inform it of a new sequence number length. Both frames are standard configuration frames in H.264. When the pass-through video ends and the Miracast session switches back to the screen UI, we execute the same

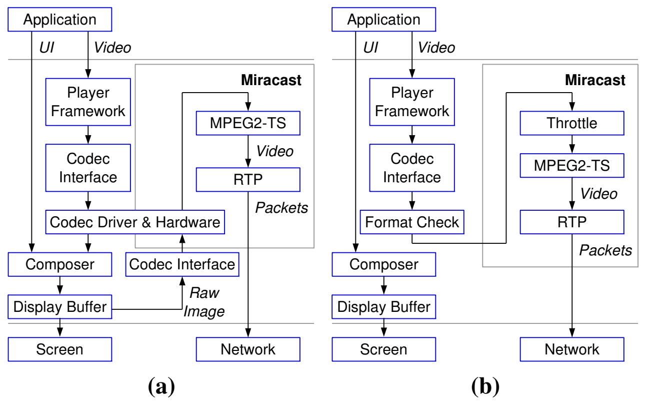

Figure 15: Work flow of video pass-through (b) in contrast to the default Miracast (a).

operation and request the codec to generate an I-frame[2] to start a new sequence of stream, without wrongly referring to frames in the pass-through stream. The overhead in delay caused by the resetting operation is minimal since most of the frames involved are without video data and are generated instantly, and the sink only needs very little extra time to process these frames.

4.3.2 Enabling Batching and Prefetching in Local Video Miracast

Video pass-through enables locally coded video data to be cast more efficiently, following two classical schemes: *(i)* Batched transmission; *(ii)* Prefetching of batched frames to the sink well before playback. Although we can enforce such schemes without video pass-through, the resulting playback latency will grow proportionally with batch size, because the codec outputs video frames at a constant pace (*e.g.*, 30fps) commensurate with the sink's playback frame-rate. For instance, our experiments show that aggregating merely 12 frames entails 1.2 seconds of waiting time (Section 5.1), apparently undesirable in practice.

To enable batching, we add a throttling function in the codec interface (Figure 15(b)), such that video frames are passed to the MPEG2-TS only if their number exceeds a batch size threshold. Meanwhile, we run a simple prefetching scheme: an entire batch of frames is sent once the due time of the first among them comes. To configure the batch size properly, we first need to understand the impact of batching/prefetching on power consumption and relevant tradeoffs.

Impact of batching on power consumption. We first cast a local video with pass-through and measure the power consumption under different batch sizes. Figure 16 plots the results. We evaluate a 2.4 GHz, 5.2 GHz channel, with and without our optimized PSM management scheme (Section 4.5.1). For all 4 settings, we observe the power consumption drop fits an exponential relation with batch size S_{batch}:

$$P_{\text{total}} = Ae^{-B\times S_{\text{batch}}} + P_0 \tag{5}$$

where the first term is the overhead and P_0 the energy truly spent on data processing/transmission. S_{batch} denotes batch size as in number of aggregated video frames. A and B are empirical parameters defining how large the initial overhead is and how quickly it dies out with frame aggregation.

With the vanilla WiFi Direct PSM, substantial overhead still exists due to a tail effect: the Miracast source keeps the network interface alive after each batch of transmission [14]. Such overhead is consistently amortized as batch size increases, leading to consistent power consumption drop. In contrast, PSMlock cuts the PSM tail, so marginal power saving is observed ($<$ 10 mW beyond a batch size of 10).

[2]In H.264, an I-frame is a self-contained frame whose encoding does not depend on other frames.

It might seem that latency will still grow with batch size (albeit more slowly than the case without video pass-through), due to the growth in batch download time, and thus it will depend on the output video bit-rate and WiFi Direct link throughput. However, in practice the player will always buffer substantial video data before the playback starts, in order to counteract the jitter effects. So the additional latency from batching may be merged with the initial buffering delay. We will quantify such latency effects in Section 5.1.

Impact of prefetching. An inherent tradeoff exists when the sink prefetches batches of video frames: if the user interrupts and stops the video, the prefetched but undisplayed frames will be wasted. Thus, the actual power saving from prefetching depends on not only batch size, but also frequency of user intervention.

Figure 17 shows this tradeoff quantitatively. We used the prior model from Figure 16 to compute power saving, and then subtract the cost from wasted transmitting, assuming user interrupts the playback periodically. The results verify that power consumption drops to a valley with batch size, but increases afterwards. The less frequently user interrupts the video display, the larger the "sweet spot" batch size is. Although user behavior is hard to predict, less than 5 seconds of holding time is not common in practice. Also, the power saving trend is similar among all user holding period larger than 5 s, as long as the batch size is not excessively large (below 20).

Based on observations from the above experiments, we can see that a batch size of 10 to 20 is sufficient to harvest the majority power saving from batching/prefetching. We thus adopt a fixed batch size of 20 in our implementation. For different devices, the suitable range of batch size may be calibrated separately following the above approach.

4.4 Energy-Aware Off-channel Miracast

WiFi-Direct allows a Miracast source to freely select wireless channels independent of any infrastructure. Our off-channel Miracast leverages this property, incorporating a model-driven, energy-aware channel selection/switching scheme.

4.4.1 Energy-Aware Off-Channel Selection

Considering the strong dependency of power consumption on network contention, the Miracast source can pick the least congested channel. Real-world wireless environment is highly dynamic and exact traffic intensity is hard to predict. However, it is feasible to distinguish two channels based on their long-term (*e.g.*, second or minute-scale) contention intensity and static power. Our off-channel selection scheme follows this principle – we require the source to quickly survey available channels, and then use a model-driven approach to rank them according to their potential for saving energy.

The power consumption model in Section 3.2.1 indicates that the only contention-dependent parameter is R_{wait}, whereas others can be obtained through factory calibration. We approximate R_{wait} in a linear relation with B_{ct}, an intermediate parameter indicating intensity of contending traffic on the same channel, *i.e.*, $R_{\text{wait}} = \alpha_1 B_{\text{ct}} + \beta_1$. The linear model is calibrated in a controlled environment, where we generate different levels of contention traffic B_{ct}, follow the approach in Section 3.2.1 to measure all parameters in Eq. (2) except R_{wait} which can be easily extrapolated. Given B_{ct} and R_{wait}, α_1 and β_1 can be easily calibrated.

In an uncontrolled environment, B_{ct} may be obtained if the source can monitor and average the ambient traffic, yet the monitoring

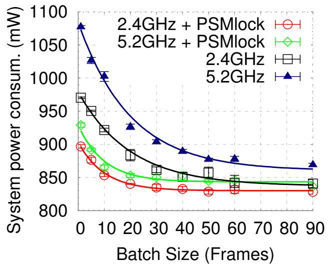

Figure 16: Power consumption vs. batch size in local video miracast.

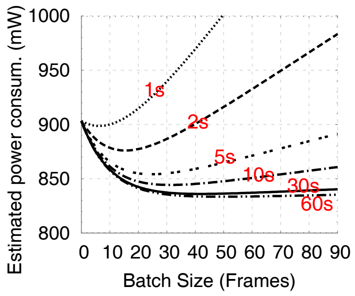

Figure 17: Power consumption vs. batch size under user interruption.

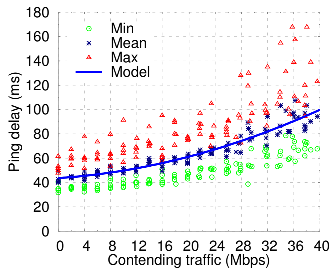

Figure 18: Ping delay vs. contention intensity B_{ct}.

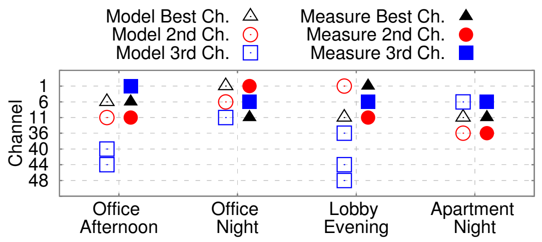

Figure 19: Effectiveness of the model-based Miracast channel ranking based on energy efficiency.

mode is unavailable on most smartphones. Instead, we require the source to send a ping message to the sink, and use a model to infer B_{ct} based on its ping delay.

To corroborate this approach, Figure 18 plots ping delay as a function of intentionally generated contending traffic. For each contending traffic setting, we repeat the measurement for 5 times, each lasting 1 second. The scattered dots depict the max, min, and mean ping delay within each 1 second interval. Despite the unpredictable variation (partly due to interfering traffic), we see that the mean ping delay is densely concentrated and fits a polynomial model $T_{\text{ping}} \approx \gamma_2 B_{\text{ct}}^2 + \alpha_2 B_{\text{ct}} + \beta_2$. The model parameters depend on the WiFi protocol version, bit-rate, and processing capability of the device, and thus can be calibrated in advance.

To summarize, the source needs to survey available channels to obtain T_{ping}, from which it obtains B_{ct} and consequently R_{wait}. Finally, it feeds R_{wait} in Eq. (2) and selects the channel with the minimum P_{static}. The dynamic power is channel independent and thus ignored.

To test the energy-aware channel selection, we place the source-sink link to multiple locations and at different time of the day. In each setting, the source pings over each channel for 1 second, obtains the T_{ping} and ranks the channels with the model. The ranking is compared with that from actual measurement, where we run a 30-second Miracast session for 3 times and then take the average power consumption. Figure 19 shows that, the model can select the top channels with high confidence. In case it errs, it only confuses the order of the top 2 channels out of a total of 7 channels, which we found to have similar power consumption. We emphasize that although the model parameters are only *calibrated once* in a controlled setting, it shows high channel selection confidence in random test environment.

Many WiFi devices support contention-based automatic channel selection (ACS) [15], which may resemble our Ping-based channel ranking approach. However, the ACS mechanism is not energy-aware and it only ranks the contention intensity. Although the current Ping-based approach is more costly than ideal, production level implementation can integrate more closely with the firmware to provide better energy-aware channel selection with less overhead.

4.4.2 *Channel Flipping for Tethered Streaming*

For the tethered streaming, the AP-source and source-sink link can operate over two orthogonal channels. The source first selects an energy efficient channel for the source-sink link using the aforementioned channel selection method, and then switches (flips) between these two channels per beacon period, for video downloading and mirroring, respectively, making itself a virtual multi-channel device. Besides the flexibility of selecting an energy efficient Miracast link, this can isolate the two links, reduce their self-contention, thereby further improving the efficiency of video delivery. Such dual-channel operation is already supported by the WiFi Direct standard and does not require any hardware modification.

However, we found such fine-grained per-beacon-period switching is disabled by the WiFi firmware on Galaxy Nexus. We thus resort to a coarse-grained emulation to approximate the benefit from channel flipping. Specifically, we found that the NIC enforces the same channel for the infrastructure and P2P interface, and is willing to change only during a reassociation. Thus, whenever channel flipping is needed, we force a reassociation on the target channel. To isolate the artifact from emulation, we ignore the reassociation cost, and replace its time/power consumption value with that of a bare channel switching in actual measurement.

4.5 Other Optimizations

Besides the above principle approaches, we enforce two other schemes that refine WiFi Direct's PSM and reduces unnecessary traffic during a Miracast session.

4.5.1 *PSMlock: Fine-grained PSM Management*

Although Wi-Fi Direct has a few built-in power management mechanisms [16], such as Notice of Absence (NoA), Opportunistic Power Saving (OppPS) and legacy PSM, there exist obstacles preventing us from utilizing them effectively. NoA and OppPS are not supported by any of the Miracast devices we are aware of. Also, together with PSM, these low-level mechanisms can not cooperate with upper level applications very efficiently. Thus, we design a PSMlock scheme to tighten the PSM schedule by leveraging the unique traffic pattern of Miracast. In a continuous video streaming application, video frames are generated following a periodic pattern (*e.g.*, 33.3ms per frame or 30fps), each frame wrapped by MPEG2-TS and converted into multiple WiFi packets. In a bursty video application (*e.g.*, slide show), such periodic pattern will remain for a few video frames after each scene change until the video tail ends (Section 4.2). Both cases end up with clusters of traffic demands followed by long/short idle period.

The WiFi PSM module is oblivious of such patterns. Modern smartphone clients widely adopt an adaptive-PSM (A-PSM) [14] protocol, that keeps the wireless interface active for extra time immediately after each burst of transmission, so as to avoid frequent mode switching [5]. As a byproduct, a PSM "tail" is introduced.

Our PSMlock scheme cuts the PSM tail after each burst of Miracast traffic, as we know deterministically when the next burst will come. PSMlock is application-transparent. It establishes a signaling channel between the MPEG2-TS module and WiFi Direct interface. Whenever a new video frame is generated, MPEG2-TS informs the network module that it will packetize a new cluster of RTP/UDP packets. Afterwards, the network module polls the packet transmission queue in the kernel associated with the Wi-Fi Direct interface at 1 ms interval, and switches the NIC into sleep mode immediately once the queue is empty (implying the transmission is finished). The switching operation requires the PSM client (Miracast source) to send a null frame containing a "PM=1" flag to inform the GO (Miracast sink) that it will sleep. The NIC is turned on again upon a new signal from MPEG2-TS.

4.5.2 Reducing Invisible Background Traffic

We have also identified two redundant background traffic sources in Android, likely caused by lack of interaction between application and the Miracast framework.

Silent background audio. First, during Miracast, the audio codec is being fed audio data constantly, no matter whether the audio is silent or not. While local speakers will be put into standby by the system during absence of useful audio, Miracast code is not notified. The additional computational power and network traffic load in casting such audio data can be substantial in bursty video casting (*e.g.*, slide show) that is not associated with any audio track. We have modified the MPEG2-TS module such that it can skip silent audio. This has not affected the non-silent audio playback at the sink side.

Background image layers. Many mobile devices' screens have a status bar indicating time, WiFi download/upload activities, battery status, as well as cellular link quality. In Android, the screen composer treats the status bar as a separate layer of pixels even if it is hidden by a full-screen video. Such behavior is desired for certain applications that need both layers. In a full-screen Miracast, the status bar is invisible, but it keeps triggering new frames and traffic periodically. We found that when displaying a still image, such hidden traffic can escalate the power consumption by multiple folds.

5. EVALUATION

In this section, we present an experimental evaluation of the energy-efficient Miracast framework. The evaluation starts with a micro-benchmark profiling of the effectiveness and cost of each individual power-optimization mechanism in Table 1, followed by a summary accounting of the contribution of each mechanism across the 5 typical Miracast applications (Section 3.1).

5.1 Micro-benchmarks

Our micro-benchmark experiments are conducted in a student office during day time. Around 5 WiFi APs coexist on our experiment channel. The Miracast system on Galaxy Nexus (Android 4.2) is used as a *baseline system*. We disable the background image layer as its artifact unnecessarily magnifies the baseline power cost (Section 4.5.2). Since Galaxy Nexus is a single-stream 802.11n device, and in our test setup the source and the sink is just a few meters apart and without blockage in between, the PHY rate remains 65Mbps most of the time.

PSMlock. We first verify the PSMlock as other mechanisms are implemented on top of it. Figure 20 plots the power saving from PSMlock. Continuous video applications leave little chance for

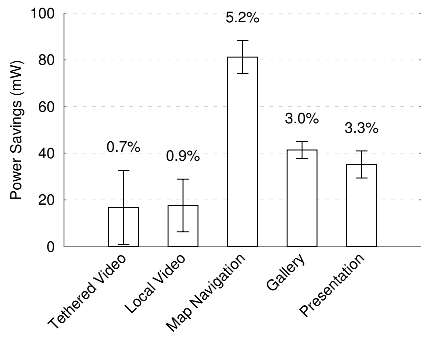

Figure 20: Power saving from PSMlock.

finer-grained PSM control (Section 4.5.1), and hence only 17.6 mW and 16.8 mW of saving on average, for local and tethered streaming, respectively. For bursty video applications, Map benefits most in terms of absolute (80 mW) power saving values, followed by Gallery and Presentation. This is consistent with the relative video traffic intensity they generate. Overall, the relative saving is 3% to 5.2%.

All our follow-on micro-benchmarks run on top of the PSMlock-optimized system.

Video Tail Cutting. We focus on the power saving from adaptive tail cutting in the local-streaming case. Tethered streaming differs only in base power. Figure 21 plots measurement results across 3 applications with bursty video traffic. Saving for presentation is only 2.3%, since the interval between video scenes is long and static system idle power dominates. Due to shorter interval, saving for the gallery application is more noticeable (5.6%). Map benefits most (19.6%) because of its short display interval and simple images that lead to more space for tail cutting.

We now evaluate the impact on video quality, focusing on the map application that experienced the most aggressive tail cutting. Figure 22 shows that PSNR is almost unaffected compared with the baseline 1-second fixed tail in Android. Thus, our adaptive tail cutting algorithm can effectively reduce video tail traffic without hurting user experience.

Video Pass-through, batching and prefetching. Since batching/prefetching needs to be enabled together with pass-through, we evaluate their joint impact on power consumption. In this experiment, the source plays a 1-minute video clip locally while miracasting. To quantify the power saving across different video configurations, the video clip has multiple versions with resolution from 144p to 720p, corresponding to bit-rate of 100 Kbps to 3 Mbps. We first measure the power saving from pass-through. Batching/prefetching's saving is measured as the additional power reduction after enabling them.

Figure 24 shows that pass-through saves 48% to 54% of power compared with the baseline system, and batching/prefetching saves additional 3% to 7% atop. It might seem counter-intuitive that the saving is insensitive to video resolution and bit-rate configurations, considering the codec power increases proportionally with video traffic load (Section 3.2). Note however that this measurement involves both codec and network power consumption. For a low resolution video, whereas pass-through saves less codec power, it saves more network power since much less video traffic is passed to the NIC. Without pass-through, the Miracast codec outputs at a constant bit-rate irrespective of the input video resolution.

To quantify the latency due to batching, we run a stopwatch application on the source and cast its UI on the sink. The timer on the sink's screen always lags behind the sources', and the difference equals Miracast's end-to-end latency. This measurement approach

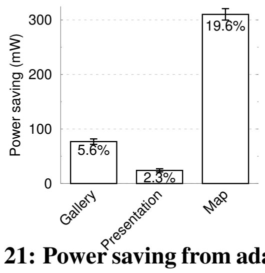

Figure 21: Power saving from adaptive tail cutting.

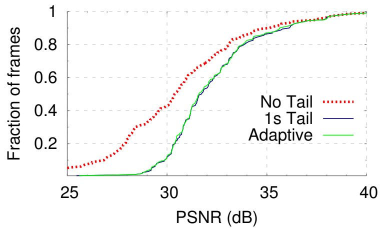

Figure 22: Distribution of PSNR across tiles in the Map application.

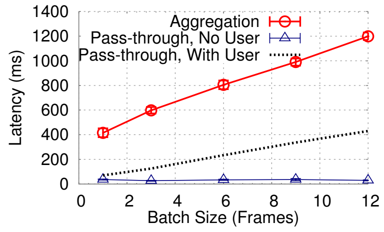

Figure 23: Impact of batching/aggregation on latency.

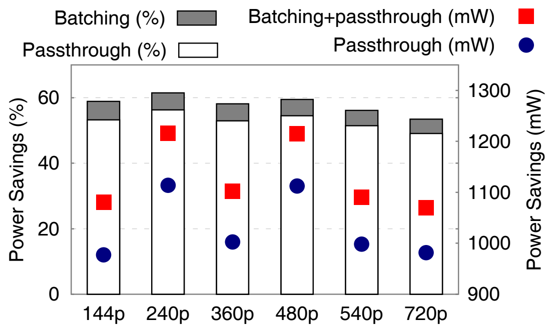

Figure 24: Power saving from pass-through and batching/prefetching (batch size 20) for local video Miracast.

does not work when pass-through is enabled, since the source does not display the video. We thus synchronize the source and sink by synchronizing them to the same NTP server prior to the start of each test, and then measure the time it takes from user's playback command to the time when the sink starts playback. Similarly, when user intervention occurs, the latency equals the time it takes from the user's stop command to the time the sink stops playback. The error introduced by NTP is measured to be less than 10ms, and keeps consistent over a period of 10 minutes, which is long enough for each test to be done.

Using a similar measurement approach as in batching, we can quantify the latency of the straightforward *aggregation* approach (Sec. 4.3.2), which waits for the codec output and aggregates video frames together. Figure 23 shows that, without pass-through, straightforward aggregation increases latency linearly. Even an aggregation of 3 video frames results in an unacceptable latency of 1 second. Note that even without aggregation (aggregation size = 1) the baseline system experiences 400 ms latency, due to the codec's processing overhead and associated queuing delay.

Remarkably, with pass-through, the latency is reduced from 400 ms to around 26 ms with a batching/prefetching size of 1 frame. As the batch size increases, the increase of air-time cost is negligible, and further absorbed by the buffering at the sink side. Even with user intervention, we observe only 400 ms of latency when 12 frames are batched.

Off-channel Miracast. We verify the model-based channel selection (Section 4.4.1) directly in the office environment with natural contending traffic. Our energy-aware approach is compared with the most power consuming channel, and a random selection that picks 3 channels arbitrarily and take the average power consumption. The Miracast link setup is the same as in Section 4.4.1. Table 2 lists the measured power consumption. For local video, our approach further saves 49 mW (2.3%) and 36 mW (1.8%), compared with worst and random case, respectively. The small percentage is mainly attributed to the low contending traffic during the test period (below 1 Mbps). Since power consumption increases linearly with contention traffic,

	Energy-aware	Worst-case	Random
Local (mW)	1955.5	2004.6	1991.1
Tethered (mW)	2424.4	2440.5	2427.9

Table 2: Effectiveness of energy-aware channel selection. Tests done in an office environment during day time.

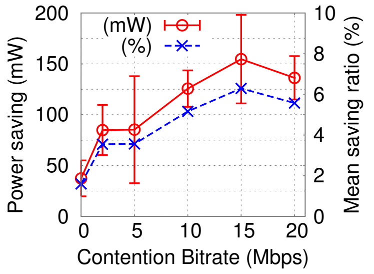

Figure 25: Power saving from off-channel Miracast for tethered video.

the saving will increase in environment where the worst-case channel is heavily congested. For tethered video, the saving is smaller due to self-contention between the AP-source and source-sink links.

We further evaluate the coarse-grained channel flipping protocol by injecting different levels of contention traffic on the AP-source channel, and allowing the source-sink link to flexibly select/switch channels. Experimental results in Figure 25 show that the saving from channel flipping increases with contention intensity, and plateaus at 8% when the AP-source channel becomes saturated by contention. Even if no external contention exists, channel flipping can still save 37 mW (1.3%) owing to reduction of self-contention.

5.2 System-Level Evaluation

We present a comprehensive evaluation of our energy-efficient Miracast system in two different contention environment: (i) severe contention, where we try to saturate the wireless channel by generating 42 Mbps of UDP traffic from a separate link, which makes the coexisting Miracast link extremely laggy. (ii) no contention, an environment with a completely idle 5 GHz channel. All other Miracast parameter settings are configured to the default in the baseline system (Section 3.1). We break down the power saving by incrementally enabling each optimization mechanism, in the following order: PSMlock, tail-cutting, pass-through, batching/prefetching/aggregation and background suppression. Contribution of each mechanism is the power saving incremental on top of previous ones. Since the experiment fixes on one channel, we disable the off-channel Miracast.

The results in Figure 26 show that, compared to the baseline system, the *overall power saving* ranges from 29% to 61% depending on application traffic patterns. Notably, contention intensity plays a

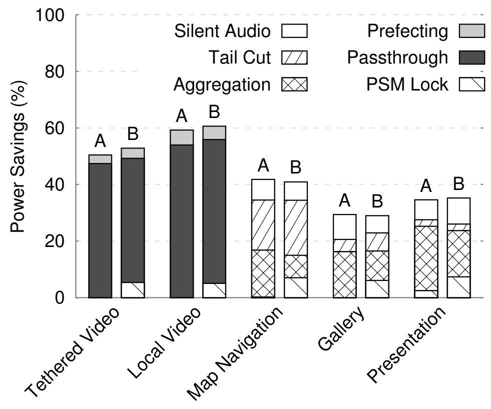

Figure 26: Power saving breakdown for different applications, in two contention environment. A: under severe contention. B: without contention.

marginal role on the overall percentage of power saving, although it affects the absolute value of power consumption by up to 100 mW (Section 3.2). However, contention does vary the relative contribution of different optimization mechanisms.

For continuous video applications, power saving comes predominantly from pass-through (42-54%), followed by batching/prefetching (3-7%). PSMlock barely helps in severe contention as each cluster of traffic is broken by contending transmissions, yet still offers 5% of saving when the channel is idle.

For bursty video applications, PSMlock exhibits similar behavior. The main contribution comes from tail cutting, frame aggregation, and background audio suppression. Tail cutting shows consistent savings as in our microbenchmark evaluation. By reducing redundant background traffic, silent audio suppression can save 5% to 9% of power. For frame aggregation, we configure the batch size to 3 video frames, which adds 200 ms of latency on top of the baseline (Section 5.1), and saves 17-21% in an idle environment, and 7% to 16% under severe contention.

We emphasize that the power saving evaluation involves an entire smartphone system, instead of the Miracast application power alone. Among all contributing factors, only the off-channel miracasting mechanism adopts a model-driven approach, and is evaluated separately in Section 5.1.

6. DISCUSSION AND FUTURE WORK

Our implementation and experiments have been conducted on the Galaxy Nexus Android phone. However, the Miracast power optimization mechanisms are not restricted to any hardware model. They do require modifications to the Miracast software inside the source node and may result in different power savings across different models.

Although Miracast is envisioned to support highly interactive applications like mobile gaming, we have not found any of such applications tailored to Miracast. Due to the 400 ms of end-to-end latency (Figure 23), we contend that the current Miracast framework is unsuitable for mobile gaming. Though pass-through and prefetching can reduce latency by an order of magnitude, they only help applications that provide encoded video frames. Improving the responsiveness/energy of interactive Miracast requires a joint optimization of codec and queuing operations between system models, which is left as our future work.

Due to limitation of the available firmware, we have not explored alternative WiFi Direct power saving protocols (*e.g.*, NoA) which, compared with PSMlock, enables more fine-grained control of the source's sleeping schedule. Similarly, the off-channel WiFi Direct, once enabled in firmware, will provide more power saving than the coarse-grained emulation approach that we evaluated. With new generation of WiFi, transmission cost may decrease. However, we note that the main power cost in Miracast lies in codec processing, plus the idle listening power caused by redundant traffic that keeps the NIC alive. Thus, we expect video pass-through, prefetching/batching and tail cutting to be useful even if WiFi evolves to much higher rate.

There are also some practical concerns for production level implementation of the video pass-through mechanism. For example, some users or applications may wish to pass-through full screen videos only, thus it is possible for other elements on the screen to be displayed together with the video side by side on the sink. We should allow users and applications to control their preferences about the behavior. Also, it is possible that multiple video elements can be present simultaneously while Miracast is in use. In this case, it might be favorable to disable video pass-through temporarily, or to employ extra algorithm to determine which video is the "main" one and should enforce pass-through.

7. RELATED WORK

Quality-aware mobile video streaming. Echoing the fast penetration of mobile devices in the past decade, substantial research work has focused optimizing video *quality* over wireless networks. The overarching goals involve minimizing startup latency, containing jitter under bandwidth fluctuation, *etc.* [17]. To date, commercial deployment has adopted many solution mechanisms [18], *e.g.*, video bit-rate adaptation, caching [19], and Dynamic Streaming over HTTP (DASH). The same server may enforce different mechanisms over different mobile devices or applications to balance between a matrix of practical factors such as video traffic profile and network condition.

Energy-aware mobile video streaming. Apart from performance optimization, energy cost reduction has been a parallel theme. Most solutions shape the downlink video traffic to create intermittent sleeping opportunities for the mobile client, while respecting video quality constraints. Depending on the vantage point of execution, the solutions can be classified as follows. *(i) Client-side protocols* allow a mobile video receiver to proactively pull contents from the streaming server based on playback buffer status [20] or prediction of demand [21]. *(ii) Proxy-based protocols* act on behalf of the server to reshape video traffic into bursts, and harnesses WiFi's PSM for clients' sleep scheduling [22]. *(iii) Server-assisted protocols* customize the traffic scheduler at streaming servers. For instance, it has been observed that YouTube, Vimeo and DailyMotion servers trigger a chunk-mode strategy [18] to create periodic bursts. Ultimately, the power saving depends on how aggressive the client's PSM decides to harnesses the sleeping opportunities.

At the application layer, content adaptation offers a flexible means of trading video quality for energy saving. The H.264 scalable video coding standard [23] compresses video data into a low-rate base-layer plus enhancement layers that can be gracefully added. It allows a client to inform the server of its video quality choice according to decoding capability and power budget [24]. The server is ready for such customized requests when running DASH [25].

System-level power optimization for mobile video. From a mobile system perspective, a variety of strategies exist to optimize the power consumption, some applicable beyond mobile video playback. For software codecs, dynamic voltage/frequency scaling can be executed based on observation of video workload [26]. For ASIC codecs, power consumption is closely related with the computational cost in compression. By modeling the relation between computational cost and rate-distortion (quality reduction), and leveraging extraneous information (*e.g.*, motion prediction from sensors [27]), it is possible to customize a set of codec parameters to optimize the power-quality tradeoff [28]. Yet to our knowledge, such customization is not supported by commercial smartphone video codecs. In addition, adaptive algorithms have been proposed that change backlight level or image brightness [29, 30], to curtail the huge display power cost during video playback. On the other hand, caution has been posed *w.r.t.* fidelity loss due to aggressive screen content-aware optimization [31].

Network interface power optimization. Many energy saving protocols that are application-independent have been proposed to address the inherent limitations of the WiFi PSM. For example, NAPman [32] and SleepWell [33] isolate the traffic bursts of different APs/clients, to reduce the idle listening power due to contention. Many COTS smartphones use A-PSM (Section 4.5.1), but it is shown to nullify the energy saving of PSM during video streaming, as it tends to keep the client in active mode [34]. Further improvement to has been proposed that cuts A-PSM's keep-alive or tail time [14]. In cellular networks, the mobile network interface also bears a keep-alive period after each transmission, which incurs substantial *tail energy* when traffic is bursty [35]. The video tail problem that we have identified bears similar spirit, but a different tradeoff – between energy saving and video quality rather than network performance.

New protocols have also been designed to optimize the power consumption of mobile hotspots. DozyAP [36], for example, introduces a coarse sleep scheduling mechanism to 802.11 softAP. Cool-Tether [37] aggregates the cellular connections from multiple smartphones as a single backhaul, and employs a reverse-tethering architecture, requesting the smartphones to act as clients of a mobile host, thus enabling their PSM. The latest WiFi-direct power-saving schemes may fundamentally simplify such protocols.

Optimizing screen sharing applications. Screen sharing is a relatively new mobile application that garnered little attention in the research literature. Most recent work [38,39] focused on maintaining video quality. Ha *et al.* [40] simulated a frame-rate adaptation scheme that reduces Miracast traffic load by analyzing the dynamism of screen content. This approach involves substantial computational overhead and the practical power saving remains unknown.

Unlike Miracast, Chromecast [41] allows the sink side to directly download videos from the Internet, at the cost that each App must be modified accordingly, whereas Miracast provides a application-transparent solution. Also, in some situations the sink might not have Internet access, or the source has access to a private network while the sink does not, making it impossible for the Chromecast approach to work properly. Thus, the Miracast approach is more universal.

8. CONCLUSION

In this paper, we have conducted the first systematic measurement study on the power consumption of Miracast. The measurement enables a modeling framework that associates network/codec power with the Miracast channel selection as well as video traffic load. Insights from the measurement lead to a set of optimization mechanisms which uncover and remove redundant Miracast traffic, curtail unnecessary computations, and improve transmission efficiency via batching/prefetching. Our implementation on a Miracast-compatible smartphone consolidates these mechanisms and demonstrates substantial power savings.

Acknowledgement

We appreciate insightful comments and feedback from the shepherd, Romit Roy Choudhury, and the anonymous reviewers. The work reported in this paper was supported in part by the NSF under Grant CNS-1318292, CNS-1343363, CNS-1350039 and CNS-1404613.

9. REFERENCES

[1] Android Nexus Device, "Project Your Android Device's Screen," 2014. [Online]. Available: https://support.google.com/nexus/answer/2865484

[2] Microsoft, "Project to a wireless display with Miracast," http://windows.microsoft.com/en-us/windows-8/project-wireless-screen-miracast, 2014.

[3] ——, "Microsoft Wireless Display Adapter," http://www.microsoft.com/hardware/en-us/p/wireless-display-adapter, 2014.

[4] Amazon.com, Inc., "Use Amazon Fire TV as a Display Mirroring Destination," 2014. [Online]. Available: http://www.amazon.com/gp/help/customer/display.html?nodeId=201453020

[5] M. Kennedy, A. Ksentini, Y. Hadjadj-Aoul, and G. Muntean, "Adaptive Energy Optimization in Multimedia-Centric Wireless Devices: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, 2013.

[6] D. Camps-Mur, X. Pérez-Costa, and S. Sallent-Ribes, "Designing Energy Efficient Access Points with Wi-Fi Direct," *Computer Networks*, vol. 55, no. 13, 2011.

[7] Wi-Fi Alliance, "Wi-Fi Display Technical Specification v1.2," 2011.

[8] Richard Lai, "CarKarPlay Display Mirrors Your Smartphone on Your Dashboard," 2014. [Online]. Available: http://www.engadget.com/2014/08/22/ipazzport-carkarplay-wireless-display/

[9] Monsoon Solutions, Inc., "Monsoon Power Monitor," http://www.msoon.com/LabEquipment/PowerMonitor/.

[10] A. Garcia-Saavedra, P. Serrano, A. Banchs, and G. Bianchi, "Energy Consumption Anatomy of 802.11 Devices and Its Implication on Modeling and Design," in *Proc. of CoNEXT*, 2012.

[11] M. Carvalho, C. Margi, K. Obraczka, and J. Garcia-Luna-Aceves, "Modeling Energy Consumption in Single-hop IEEE 802.11 Ad Hoc Networks," in *Proc. of IEEE ICCCN*, 2004.

[12] L. Zappaterra and H.-A. Choi, "Efficient Power-aware Multi-Channel Selection for Dynamic Cognitive Radio Networks," in *Proc. of IEE WCNC*.

[13] B. Yang, Y. Shen, X. Guan, and W. Wang, "Energy-aware Opportunistic Channel Access With Decentralized Channel State Information," in *Proc. of IEEE CDC*, 2011.

[14] A. J. Pyles, X. Qi, G. Zhou, M. Keally, and X. Liu, "SAPSM: Smart Adaptive 802.11 PSM for Smartphones," in *Proc. of ACM UbiComp*, 2012.

[15] Linux Wireless, "Automatic Channel Selection," 2014. [Online]. Available: http://wireless.wiki.kernel.org/en/users/documentation/acs

[16] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano, "Device-to-Device Communications With Wi-Fi Direct: Overview and Experimentation," *IEEE Wireless Communications*, vol. 20, no. 3, 2013.

[17] L. Guo, E. Tan, S. Chen, Z. Xiao, O. Spatscheck, and X. Zhang, "Delving into Internet Streaming Media Delivery: A Quality and Resource Utilization Perspective," in *Proc. of ACM IMC*, 2006.

[18] M. Hoque, M. Siekkinen, J. Nurminen, and M. Aalto, "Dissecting Mobile Video Services: An Energy Consumption Perspective," in *Proc. of IEEE WoWMoM*, 2013.

[19] S.-H. Shen and A. Akella, "An Information-aware QoE-centric Mobile Video Cache," in *Proc. of ACM MobiCom*, 2013.

[20] D. Bertozzi, L. Benini, and B. Ricco, "Power Aware Network Interface Management for Streaming Multimedia," in *Proc. of IEEE WCNC*, 2002.

[21] Y. Wei, S. M. Bhandarkar, and S. Chandra, "A Client-side Statistical Prediction Scheme for Energy Aware Multimedia Data Streaming," *IEEE Trans. on Multimedia*, vol. 8, no. 4, 2006.

[22] S. Chandra and A. Vahdat, "Application-Specific Network Management for Energy-Aware Streaming of Popular Multimedia Formats," in *Proc. of USENIX ATC*, 2002.

[23] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the Scalable Video Coding Extension of the H.264/AVC Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, 2007.

[24] K. Choi, K. Kim, and M. Pedram, "Energy-Aware MPEG-4 FGS Streaming," in *Proc. of Design Automation Conference (DAC)*, 2003.

[25] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An Experimental Evaluation of Rate-adaptation Algorithms in Adaptive Streaming over HTTP," in *Proc. of ACM Multimedia*, 2011.

[26] Z. Cao, B. Foo, L. He, and M. van der Schaar, "Optimality and Improvement of Dynamic Voltage Scaling Algorithms for Multimedia Applications," *IEEE Transactions on Circuits and Systems*, vol. 57, no. 3, 2010.

[27] X. Chen, Z. Zhao, A. Rahmati, Y. Wang, and L. Zhong, "SaVE: Sensor-assisted Motion Estimation for Efficient H.264/AVC Video Encoding," in *Proc. of ACM Multimedia*, 2009.

[28] E. Akyol and M. van der Schaar, "Compression-Aware Energy Optimization for Video Decoding Systems With Passive Power," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 9, 2008.

[29] P.-C. Hsiu, C.-H. Lin, and C.-K. Hsieh, "Dynamic Backlight Scaling Optimization for Mobile Streaming Applications," in *Proc. of IEEE ISLPED*, 2011.

[30] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan, "Adaptive Display Power Management for Mobile Games," in *Proc. of ACM MobiSys*, 2011.

[31] S. Chandra, J. T. Biehl, J. Boreczky, S. Carter, and L. A. Rowe, "Understanding Screen Contents for Building a High Performance, Real Time Screen Sharing System," in *Proc. of ACM Multimedia*, 2012.

[32] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu, "NAPman: Network-assisted Power Management for WiFi Devices," in *Proc. of ACM MobiSys*, 2010.

[33] J. Manweiler and R. Roy Choudhury, "Avoiding the Rush Hours: WiFi Energy Management via Traffic Isolation," in *Proc. of ACM MobiSys*, 2011.

[34] E. Tan, L. Guo, S. Chen, and X. Zhang, "PSM-throttling: Minimizing Energy Consumption for Bulk Data Communications in WLANs," in *IEEE International Conference on Network Protocols*, 2007.

[35] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "Characterizing Radio Resource Allocation for 3G Networks," in *Proc. of ACM IMC*, 2010.

[36] H. Han, Y. Liu, G. Shen, Y. Zhang, and Q. Li, "DozyAP: Power-efficient Wi-Fi Tethering," in *Proc. of ACM MobiSys*, 2012.

[37] A. Sharma, V. Navda, R. Ramjee, V. N. Padmanabhan, and E. M. Belding, "Cool-Tether: Energy Efficient On-the-Fly WiFi Hot-spots Using Mobile Phones," in *Proc. of ACM CoNEXT*, 2009.

[38] C.-F. Hsu, D.-Y. Chen, C.-Y. Huang, C.-H. Hsu, and K.-T. Chen, "Screencast in the Wild: Performance and Limitations," in *Proc. of ACM Multimedia*, 2014.

[39] S. Chandra, J. Boreczky, and L. A. Rowe, "High Performance Many-to-many Intranet Screen Sharing with DisplayCast," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 10, no. 2, 2014.

[40] J. Ha, P. Bae, K.-W. Lim, J. Ko, and Y.-B. Ko, "Poster abstract: Mobile contents on the big screen: Adaptive frame filtering for mobile device screen sharing," in *Proc. of ACM SenSys*, 2014.

[41] B. Johnson, "How Chromecast Works," 2014. [Online]. Available: http://electronics.howstuffworks.com/chromecast.htm