

LPD: Low Power Display Mechanism for Mobile and Wearable Devices

MyungJoo Ham, Inki Dae, and Chanwoo Choi, ***Samsung Electronics***

https://www.usenix.org/conference/atc15/technical-session/presentation/ham_lpd

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225



Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

LPD: Low Power Display Mechanism for Mobile and Wearable Devices

MyungJoo Ham
Frontier Computer Science Lab,
Software R&D Center,
Samsung Electronics

Inki Dae Chanwoo Choi
Software Platform Team,
Software R&D Center,
Samsung Electronics

{myungjoo.ham, inki.dae, cw00.choi}@samsung.com

Abstract

A plethora of mobile devices such as smartphones, wearables, and tablets have been explosively penetrated into the market in the last decade. In battery powered mobile devices, energy is a scarce resource that should be carefully managed. A mobile device consists of many components and each of them contributes to the overall power consumption. This paper focuses on the energy conservation problem in display components, the importance of which is growing as contemporary mobile devices are equipped with higher display resolutions. Prior approaches to save energy in display units either critically deteriorate user perception or depend on additional hardware. We propose a novel display energy conservation scheme called *LPD* (Low Power Display) that preserves display quality without requiring specialized hardware. *LPD* utilizes the display update information available at the X Window system and eliminates expensive memory copies of unvaried parts. *LPD* can be directly applicable to devices based on Linux and X Windows system. Numerous experimental analyses show that *LPD* saves up to 7.87% of the total device power consumption. Several commercial products such as Samsung Gear S employ *LPD* whose source code is disclosed to the public as open-source software at `http://opensource.samsung.com` and `http://review.tizen.org`.

1 Introduction

The popularity of mobile devices such as smartphones, tablets, and smart watches is steadily increasing and their market size has grown explosively in recent years. Tetherless mobile devices use batteries as the main energy source and power is one of the scarcest resources that should be carefully managed; energy consumption is directly translated to the usability and the value of mobile products. In addition, imprudent use of energy may lead to excessive heat dissipation, which in turn, causes a safety issue of low temperature burns [13]. One easy solution for the power saving problem is to equip better and/or larger batteries in mobile devices. However, the advancements in battery technology failed to match the ever increasing functionalities and computational demands of mobile devices [17].

A mobile device consists of many components and functions each of which consumes energy. This paper deals with energy conservation in display components. As the display resolutions increase, the energy required to operate a device grows accordingly. For example, even though the physical scales of displays have not grown much bigger, resolution has increased from 800x480 to 2560x1440. The memory bandwidth increases almost ten times and so is the energy consumed. Energy conservation in memory access for display components has received less research attentions than other components such as processors and communication interfaces although display components consume significant share of energy [5].

Energy conservation schemes prone to deteriorate the performance or QoE (Quality of Experience) of devices. Because human beings are sensitive to the degradation in visual quality, vigilant attentions to preserve the original visual quality must be accompanied in designing power saving techniques for display units. Adjustments of color depth [8], brightness level [6, 11], or refresh rate [14] may significantly affect user perception such that the quality assurance team often rejects careless schemes. Of course, there are display energy saving schemes that preserve the original quality. AFBC (ARM Frame Buffer Compression) [3, 10], Transaction Elimination [4, 16], and frame buffer compression [20] are examples of such approaches. However, most of these schemes depend on specialized hardware and their applicability is quite limited.

We aim to develop a display energy conservation scheme that neither requires the addition of specialized hardware nor deteriorate the visual quality. The proposed scheme, low power display (*LPD*), does not require any hardware modifications to the traditional and popular `i80` display architecture, Intel's 8080 like command interface for display panels. *LPD* also does not deteriorate the user experiences because it conserves the true quality of every pixel.

The main idea of *LPD* is rather simple; to reduce memory accesses and data transfers by identifying the updated regions. The idea of preserving unchanged part and encoding only changed part is widely used in motion picture encoding [22] and display rendering. The problem is how to identify the updated regions. Comparing the two consecutive frame buffers directly requires too much energy or additional hardware. Instead of direct frame buffer comparison, we exploit the knowledge that the OS already possesses. In other words, *LPD* extends the design domain from HW-kernel to HW-kernel-middleware. In Linux and Tizen, a window system (X Server) and a compositing window manager (Enlightenment in the case of Tizen) know the changed regions. LPD accesses changed regions only and transfers the retrieved regions to the display controller and display panel. Therefore, LPD reduces the memory bandwidth as well as bus utilization which in turn reduces power consumption.

LPD also has the potential to enhance the performance of other functions because *LPD* reduces main memory bandwidth and the saved bandwidth can be distributed to other memory hungry functions. Unlike the previous schemes, the computation overhead of *LPD* is minimal; it requires a few simple integer arithmetic instructions without any loops or complex computation. Finally, *LPD* is orthogonal to other display power saving mechanisms [4, 8, 11, 14, 16] such that LPD can be applied with these methods.

To reconstruct a whole display image from updated regions only, the display panel should have an internal RAM that stores the previous frame. Such a feature is commonly available in mobile devices; `i80`, one of the de facto standard display interfaces supports an internal RAM. We confirmed that many mobile devices such as Galaxy S4 and Galaxy Note 3 use the `i80` interface.

We implemented *LPD* and *LPD* has been embedded in commercial products. An earlier version of *LPD* has been shipped with Gear 2. Field tests with real products under real-world use scenarios showed that *LPD* reduce up to 7.87% of the total device power consumption when 1% of frame is updated. Full capability of *LPD* has been implemented and embedded to Gear S. We disclose the source code of full *LPD* implementation to the public at `http://tizen.org` and `http://opensource.samsung.com`. The source code is under the GPL license as a feature of Direct Rendering Manager (DRM), which significantly lessens the maintenance and porting cost for further deployment.

The main contribution of this paper is as follows:

- Improve energy efficiency of display device components that were not properly addressed while
  - preserving the transparency of applications,
  - maintaining traditional hardware architectures,
  - minimizing changes to the operating systems,
  - limiting the overhead to virtually non-existing,
  - not deteriorating the quality of pixels,
  - and allowing most of previous display power optimizing schemes orthogonally coexisting.
- The proposed scheme is fully developed and released as open source software in commercial products.

This paper is organized as follows. The next section presents the related work of display power saving. Section 3 explains the hardware architecture and the rationale of *LPD*. Section 4 shows the design and implementation detail of *LPD*. Section 5 describes the experiments and their results. Section 6 discusses follow-up research that may further enhance *LPD*. Section 7 concludes the paper.

2 Related Work

Several researchers have attacked the power consumption of display-related device components. In this section, we introduce their work and we show why we still need a new mechanism.

Adjust color depth: Choi et al. [8] have suggested a display power saving mechanism that dynamically alters color depth according to the color distribution of a frame buffer. This method scans the whole frame buffer, which usually is performed by an additional hardware to avoid excessive CPU overhead and power consumption. The mechanism is especially effective with high quality high resolution displays while it inevitably deteriorates the picture quality.

Dynamic backlight brightness: backlight is the dominant power consumption source in display systems and several backlight reduction mechanisms have been

devised [1, 6, 7, 11, 19]. Backlight reduction should be accompanied with careful pixel color adjustment to keep the fidelity of images. For example, if a frame is filled with dark pixels, we may reduce the backlight brightness while compensate the gamma values of pixels to brighter colors. Enhancing such approaches further, [21] suggested to partition a screen into multiple regions with separated backlights and adjust the backlights and colors independently for each block for extra power saving.

Dynamic backlight reduction schemes have limitations. Chang et al. [6] sacrificed brightest pixels to reduce the backlight brightness. This optimization degraded the picture quality significantly such that the degradation can be detected by naked eyes. Backlight reduction schemes also require additional full scan of each frame buffer. Full frame scan inevitably provokes additional memory transactions and power consumption. LCD (Liquid Crystal Display) where the responses of each color to brightness are non-linear spawns another complicated control problem [1]. Significant latency increment is another roadblock for the adoption of the technique [6] to latency critical applications such as games, screen scrolling, and typing [19]. Most critically, the brightness control schemes cannot be applied to AMOLED (Active-Matrix Organic Light-Emitting Diode). AMOLED displays, dispense with backlights, are considered to be energy efficient and more suitable for mobile devices [12]. A similar approach for AMOLED displays [18], which tries to adjust pixel colors, may consume much energy due to the physical characteristics of AMOLED; if a pixel changes its color too drastically in a short time, this causes much energy consumption to drive the pixel.

Dynamic display refresh rate: Kim et al. [14] have suggested to dynamically scale the refresh rate of displays. We have applied the technique as a device driver of DVFS framework (devfreq) in the Linux kernel [9], but failed to meet the requirement of picture quality maintained by our quality assurance teams. With further optimizations, the techniques can be effective and applied with *LPD* orthogonally.

Compression: another approach is the frame buffer compression [3, 10, 20]. Compression reduces data size and thus decreases bus traffic and memory operations. Compression is usually performed by an additional non-standard hardware because compressing the whole frame buffer for every frame incurs heavy computational overheads [20]. Compression also incurs power consumption; even with a dedicated FPGA based hardware [20], compressing and decompressing

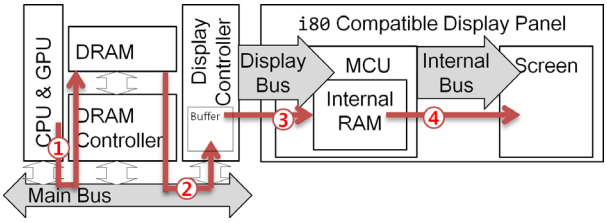


Figure 1: i80 Display Hardware Architecture

frame buffers of 640x480 with 18 bit color depth has consumed additional 30 mW. ARM's Adaptive Scalable Texture Compression (ASTC) [2, 15] provides higher compression rate than other conventional frame buffer compression mechanisms. However, ASTC is limited to textures for GPUs and uses lossy compression mechanisms.

Skip duplicated transmissions: the prior methods that are most similar to *LPD* are the mechanisms that skip transmissions of duplicated parts. Whelan et al. [24] saves the whole frame buffer at the display controller and allows skipping the transfer of a new frame from the main memory to the frame buffer if there are no changes. The benefit of skipping is achievable only when there is not even a pixel of change in a frame [24]. We implemented a variant of this scheme in Samsung Gear series products. In this implementation, we can turn the whole CPU off (suspend-to-RAM) along with the display controller while the screen kept on.

Another similar approach is Transaction Elimination developed by ARM [4, 16]. Transaction Elimination allows a GPU to skip transmitting unchanged parts of its frame buffers to the main memory based on CRC signatures. This approach requires ARM's Midgard GPU architecture. Transaction Elimination reduces data transfer to the main memory only maintaining the data transfers from the main memory to the display panel via the display controller. On the contrary, *LPD* can reduce the data transfers from the main memory to the display panel and does not require using a specific GPU.

3 Background

Figure 1 shows the hardware architecture and *LPD* procedure. Arrows with circled numbers represent image data transmission between hardware components. *LPD* requires a display panel with the i80 display interface and a display controller supporting "partial mode". In the partial mode, the display controller fetches a rectangular subset of the frame buffer from the DRAM to its buffer (step 2). The rectangular subset



(a) Current frame (b) Next frame (c) Updated region

Figure 2: Example of a Display Content Change

should contain all updated parts.

Figure 2 shows an example of a rectangular subset of updated parts. Let us assume that the display content is updated from the current frame (Figure 2(a)) to the next frame (Figure 2(b)). In the example, there are two updated components; the red second hand and the alarm shown on the upper right corner. These updated parts are represented by two light gray boxes in Figure 2(c) and the rectangular subset, outlined by a dotted blue rectangle, is a larger box that contains the two updated components.

After user programs draw images with CPU and GPU (step 1) on the frame buffer in the main memory (DRAM), graphical middleware, X Window and Composite Manager, process the raw image and send the processed data to the kernel. Referring the processed data, the kernel along with related device drivers configures the display controller. Next, the display controller read the updated part from DRAM to its buffer (step 2). The display controller transfers the updated part to the internal RAM of the display panel via a hardware-to-hardware line called "display bus" (step 3). Finally, the display panel lays out the contents in the internal RAM on the screen. *LPD* enhances step 2 and step 3 procedures. Step 2 involves with main memory read, transfer on the main bus, and write into the buffer in the display controller. Step 3 consists of buffer read, transfer on the display bus, and write into the internal memory. Note that the final transmission to the screen (step 4) contains the whole frame buffer and is not reduced by *LPD*.

3.1 Simple Analysis of Expected Power Saving

In this section, we describe the rationale that led us to the design of *LPD* with a simple analysis of the expected power saving. Let P_u ($0 \leq P_u \leq 1$) be the proportion of the updated rectangle to the whole frame. Also, let f be the frame rate and S be the size of a whole frame, which is usually the product of width, height, and color depth. Then, T_L, the memory bandwidth that *LPD* consumes to transfer the updated rectangle from the main memory to the internal RAM through the display controller, is

$$T_L = P_u \cdot S \cdot f \tag{1}$$

The traffic without *LPD*, T_0, between the same components is:

$$T_0 = S \cdot f \tag{2}$$

Note that the bandwidth of DRAM read and main bus transmission, display bus transmission and internal RAM write are the same because we assume no compression or modifications in the transmission chain from the DRAM to the internal RAM. *LPD* is orthogonal to such operations and any benefits obtained by compression can be equally applied to *LPD* as well as to non-*LPD* schemes.

The updated contents should be readily available in the DRAM when the display controller accesses the DRAM because the controller is not aware of processor caches; i.e., there is no cache coherency support between CPU and controller. It also means that caches of processors cannot be involved and every bit read, moved, or written with the display controller or the display panel is a direct memory-to-device or device-to-device operation. Therefore, we can assume that the power consumed in memory read, transfers on the main bus, and transfers on the display bus are not affected by caching. P_L and P_0, the power consumed by *LPD* and non-*LPD* schemes, respectively, are given as

$$\begin{aligned} P_L &= C \cdot (T_L) \\ P_0 &= C \cdot (T_0) \end{aligned} \tag{3}$$

, where C is a coefficient representing the sum of the energy consumption rates of all involved operations. The total power saved by *LPD*, P_{save} is:

$$\begin{aligned} P_{save} &= P_0 - P_L \\ &= C \cdot T_0 - C \cdot T_L \\ &= C \cdot (1 - P_u) \cdot S \cdot f \end{aligned} \tag{4}$$

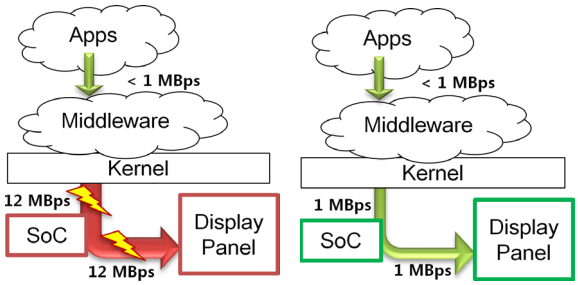
This shows that the power saving is proportional to P_u, the proportion of updated regions. As we can see in Eq. (4), *LPD* enjoys greater savings with devices with higher resolutions and higher frame rates. Note that mobile displays have undergone disruptive technology advances in the last decade and this trend may continue in near future; recent mobile phones have displays of 1920x1080 resolution or higher at 60 fps.

In Section 5, we show the effectiveness of *LPD* with a series of experiments with Samsung Gear 2. We also show how the model driven in this section fits with the experimental results.



(a) Current frame (b) Next frame (c) Torn frame

Figure 3: Screen Tearing Example



(a) Without *LPD* (b) With *LPD*

Figure 4: The Concept of *LPD*

3.2 Overhead of Brute Force Mechanisms

In this section, we analyze the potential overhead of *LPD* similar mechanisms implemented in a brute force style. Instead of using the processed information provided by middleware, these methods identify the updated regions by frame by frame comparison.

Method 1. Compare each pixel to identify updated regions. This requires reading two frames and the required memory bandwidth, M_r, is

$$M_r = S \cdot f \cdot 2 \tag{5}$$

The maximum benefit due to reduced transfer is achieved when there are no updated regions. The maximum benefit is $M_r/2$ and the overhead overwhelms the benefit.

Method 2. Compare CRC values of frame buffer blocks. This is what Transaction Elimination does [4, 16] with an additional hardware for GPU to main memory transmissions. If we perform the same operation with software, we need to read a whole frame once and should calculate CRC at the speed of memory bandwidth. The overhead still overwhelms the benefit as well.

As indicated above, brute force mechanisms that identify the differences based on frame-by-frame comparison are inappropriate. Note also that hardware-based approaches [4, 16] incur inevitable overheads of gate count, energy, and licenses.

3.3 Screen Tearing and Tearing Effect

Screen tearing may appear if the image transfer from the display controller to the internal RAM is not properly synchronized with the display refresh by the MCU. Figure 3 shows an example of screen tearing.

One scenario that causes the screen tearing of Figure 3 is as follows. While the MCU is scanning its internal RAM containing the current frame for the display refresh, the display controller transfers the next frame overwriting on to the internal RAM. If the speed of display controller transfer is faster than the display refresh, part of the internal RAM that is not displayed may be updated with the next frame. As a result, the screen shows a mix of both frames as depicted in Figure 3(c).

A display panel generates a tearing effect (TE) signal to notify the kernel that the panel has completed drawing the image from its internal RAM. A display controller should start sending the next frame to the display panel after receiving the TE signal and should complete the transmission before the MCU starts to refresh the next frame. In other words, steps shown in Figure 1 should be synchronized with the TE signal.

In Section 4.2.3, we discuss the issue of screen tearing in a greater detail. Screen tearing becomes more serious with *LPD* as the transfer latency becomes less deterministic and device drivers are required to add operations with exact timing. Section 4.2.3 describes the synchronization mechanism that *LPD* uses to mitigate the issue.

4 Design and Implementation

Figure 4 shows the design concept of *LPD*. *LPD* utilizes the information already known to applications and middleware to reduce the amount of information handled by hardware components. Suppose a device with 320x320 resolutions. With 4 bytes per pixel and 30 frames per second, the amount of information required to transfer is about 12 MB/s. If we further assume that 8% of a frame is updated on the average, then the required bandwidth for the updated regions is about 1 MB/s. Without *LPD*, the required bandwidth from the software stack to the display panel is still 12 MB/s because full frames are transferred regardless of updated regions. With *LPD*

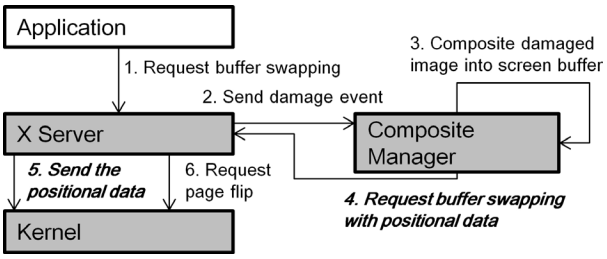


Figure 5: Interaction of Middleware and the Linux Kernel for *LPD* implementation

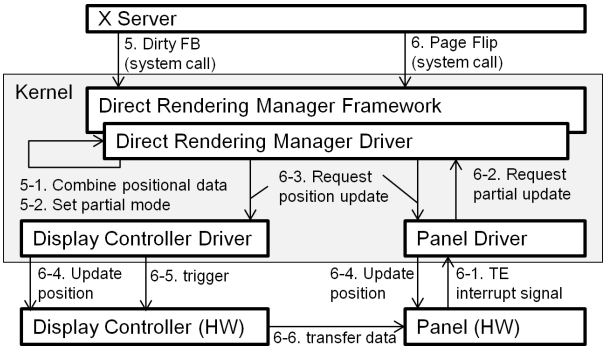


Figure 6: Interaction of Linux Kernel and Hardware

as shown in Figure 4(b), the bandwidth is reduced to 1 MB/s. A System-on-Chip (SOC) usually has processors, main memory, and a display controller. Let us examine the procedure of *LPD* from the top to the bottom and the issues we have encountered in the course of *LPD* implementation.

4.1 Userspace Middleware Interaction

Figure 5 shows how the window system (X Server) and the composite manager in userspace interact with applications and the kernel. The numbers in the interaction vectors denote the sequence of events. The shaded boxes and descriptions in italic are the components affected and interactions modified by *LPD*, respectively. Such modifications allow the kernel to have the information required to identify the updated regions. *LPD* does not require additional modifications in applications or other middleware components.

The sequence of interactions in userspace flows as follows:

1. An application requests a buffer swap to the X Server.
2. The X Server notifies a damage event to the composite manager. Each damage event contains the positional data of an updated region.
3. The composite manager composes the screen image with the damage event information provided in step 2.
4. The composite manager requests a buffer swap to the X Server. In *LPD*, this request includes positional data of updated regions. In non-*LPD*, this request does not include any information.
5. In *LPD*, the X Server transfers the positional data to the kernel with the "Dirty FB" kernel interface described in Section 4.2.1. In non-*LPD*, this step is skipped.
6. The X Server requests a page flip to the kernel so that image data can be sent to the screen with "Page Flip" interface described in Section 4.2.2.

As shown, the modification in the middleware is minimal and the backward compatibility of the modified userspace components is preserved. Because the composite manager has been already tracking the updated regions (or damaged regions in their notation) in order to optimize rendering performance, we simply modified the composite manager to report back what it already comprehends as one aggregated updated region. Then, the X Server just relays the information. With such simple and straightforward notifications, we can enjoy the benefit of reduced data bandwidth. It is worth to note that *LPD* incurs constant computational and space complexity.

4.2 Kernel Interaction

As mentioned earlier, we use two userspace-to-kernel interfaces: Dirty FB and Page Flip. The detailed in-kernel operations of the two interfaces are described in Figure 6. The Dirty FB triggers sub-steps 5-1 and 5-2 and the Page Flip interface involves with sub-steps 6-1 to 6-6. Note that some interfaces are not software-driven interactions. For example, 6-1 is an interrupt from hardware and 6-6 is a hardware-to-hardware transmission.

The two kernel interfaces, Dirty FB and Page Flip, are not new or non-standard interfaces but are standard Linux kernel interfaces that had been kept in the mainline. We also do not change the semantics of the interfaces. Note that being a standard interface is not coincident with the popular or frequent use of the interface. Both the Dirty FB interface and Page Flip interface are seldom used or not fully used.

We obeyed the syntax and semantics of Linux mainline interfaces. *LPD* is easily upstream-able and reusable

by other device drivers in various kernel versions by different vendors. The upstream-ability and the induced compatibility add yet another benefit to *LPD*: maintainability, which enables us to let the open source community maintain *LPD* along with later versions of Linux kernel and additional device drivers. We expect that we can upstream all the required pieces to the mainline Linux kernel soon.

4.2.1 Dirty FB

Dirty FB, a kernel-userspace interface, allows the X Server to send multiple sets of updated regions (rectangle forms consist of the left-top and right-bottom coordinates) to the kernel DRM driver before the X Server issues Page Flip. Without *LPD*, the X Server does not need to use the Dirty FB interface because the X Server assumes that a whole frame is updated. The operation of Dirty FB consists of the following steps as shown in Figure 6.

Step 5: The X Server sends one or multiple updated regions to the kernel DRM driver.

Step 5-1: The DRM driver merges input regions into a single rectangle that contains all updated regions. The larger box with a dotted blue outline in Figure 2(c) represents the aggregated single rectangle.

Step 5-2: The DRM driver remembers the coordinates of the aggregated update region and uses "partial mode" for the next frame transmission.

Most embedded display controllers can transfer image data of a single rectangular region to the display panels in one single transfer. For each TE interrupt signal, the display controller can conduct one transfer only and there is only one TE interrupt signal per display refresh. Therefore, in order to avoid image quality deterioration due to frame drops, *LPD* should combine multiple updated regions into one.

Let the left-top coordinate and the right-bottom coordinate of each updated region be $L = (L_x, L_y)$ and $R = (R_x, R_y)$, respectively. Each updated region can be expressed by a pair of L and R. Then, L' and R', the left-top and the right-bottom coordinates of the aggregated updated region covering n updated regions are derived as:

$$\begin{aligned} L' &= (min(L_{x1}, \ldots, L_{xn}),\ min(L_{y1}, \ldots, L_{yn})) \\ R' &= (max(R_{x1}, \ldots, R_{xn}), max(R_{y1}, \ldots, R_{yn})) \end{aligned} \quad (6)$$

, where $L_i = (L_{xi}, L_{yi})$ and $R_i = (R_{xi}, R_{yi})$ are the left-top and the right-bottom coordinates of the i-th updated region.

4.2.2 Page Flip

In non-*LPD*, the window system requests a frame buffer change via the Page Flip interface. An invocation of Page Flip updates the memory address to the requested frame buffer of the display controller hardware. Then, the display controller may access the requested frame buffer by setting a trigger bit after a TE signal is issued.

If the display controller is in a partial mode (*LPD* enabled), the Page Flip behavior is slightly different because we cannot simply switch frame buffers for each frame. Instead of transferring the whole frame buffer, the controller transfers the updated region only. In the partial mode, configured by *LPD*, a Page Flip request updates the relevant registers (sub-step 6-4) that include the memory base and the offset address to the updated region, start and end positions of the overlay, and the line size. Note that the partial mode does not require the display controller to support input/output memory management unit (IOMMU). The partial mode only requires the controller to access a rectangular subpart of a frame buffer. It does not depend on whether the frame buffer is in a physically contiguous memory chunk (conventional DMA) or in a virtually contiguous memory chunk (DMA with IOMMU).

In the partial mode, like the Page Flip request, a TE interrupt signal (sub-step 6-1) initiates the update of MCU registers that includes the start and end coordinates of the internal RAM. Note that a Page Flip request activates *LPD* if Dirty FB has been called after the previous Page Flip request. Otherwise, the kernel DRM subsystem assumes that the user wants to replace the whole contents.

As shown in Figure 6 with the sub-steps from 6-1 to 6-4, the TE interrupt (6-1) allows the panel driver to request a partial update to the kernel DRM driver (sub-step 6-2). Then, the kernel DRM driver requests position updates to both display control driver and panel driver (sub-step 6-3) that commonly are sub device drivers of the DRM driver. Then, these two sub drivers update positional information of their corresponding hardware (sub-step 6-4). Lastly, the display controller driver commands the display controller (sub-step 6-5) to initiate the data transfer (sub-step 6-6).

4.2.3 Prevention of Screen Tearing

While implementing *LPD* on experimental devices, we have experienced screen tearing. Without *LPD*, because frame data transmission times are long and deterministic, careful manipulation of the display controller is not required and the tearing is not an issue. A mechanism to

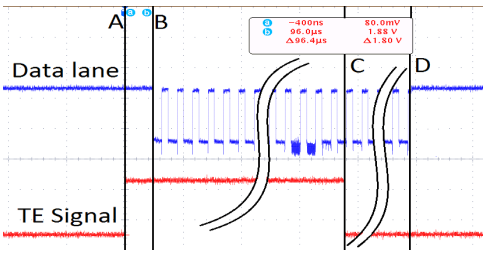


Figure 7: Data and TE Signals

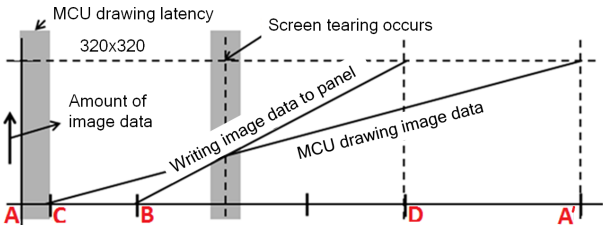


Figure 8: Screen Tearing with Faster Image Data Transfer

prevent screen tearing under varying frame data transfer latency has been implemented and included in *LPD*.

LPD uses the TE signal and its handler to prevent screen tearing. Figure 7 shows the timing of data lane signal of the display bus controller and the TE interrupt signal of the display panel. The TE signal notifies when to transfer the next frame.

When the TE signal occurs at point A in Figure 7, the MCU of the display panel has completed drawing the contents of its internal RAM to the screen. About 96 μs later, at point B, the display controller initiates the data transfer to the display panel. Point C denotes the time when the MCU of the display panel has completed drawing the contents. Point D denotes the time when the display controller has completed writing the contents to the display panel.

LPD classifies the events that cause screen tearing into two classes.

- Case 1. The display controller speed is slower than the drawing speed of the MCU.
- Case 2. The display controller speed is faster than the drawing speed and an image data transfer (at point B) starts while drawing the previous frame (point C already occurred). This case is depicted in Figure 8. The markers (A to D) in both figures denote the same types of events. A' denotes the next A event.

In order to prevent the first case, *LPD* completes configuring every related device between A and B and sets the display controller faster than the drawing speed of MCU. In order to prevent the second case, *LPD* ensures that B starts after A and before C.

Another issue with *LPD* arises when multiple hardware overlays are applied. Samsung Gear 2 supports up to five overlays although it mostly uses only one. If we use multiple hardware overlays simultaneously, the display controller sends a merged image from multiple virtual frame buffers (hardware overlays) to the panel. The current implementation of *LPD* does not support aggregate updated regions across multiple hardware overlays. Therefore, if multiple hardware overlays are used, the transfer mode should be fixed to full screen mode before the display controller starts to transfer image data to the display panel. *LPD* configures the transfer mode to partial mode (*LPD* enabled) if a single hardware overlay is used and configures to full screen mode if multiple overlays are used. *LPD* checks if the partial mode may be enabled or the full screen mode should be enabled based on the Page Flip request. In order to support multiple hardware overlays, *LPD* should be updated to track the origin point of each hardware overlay.

5 Experiments

We have examined the functionality and performance of *LPD* by conducting experiments on Samsung Gear 2. The hardware specifications of Gear 2 are as follows:

- Display type & size: AMOLED, 1.63 inch
- Resolution: 320x320
- Frame rate: 30 FPS
- Application Processor (SoC): Exynos 3250
  - CPU: Dual ARM Cortex A7 1.0 GHz
  - GPU: Mali-400 MP
  - Main Memory: 512 MiB LPDDR3 DRAM

We have conducted two different sets of experiments. The first set of experiments is performed with a synthetic power consumption benchmark; a benchmark application runs directly on the Linux kernel without the X server window system. In the first set of experiments, we varied the size of updated regions.

The second set of experiments involved with publicly released Tizen wearable applications: W-launcher, Heart Rate, Setup-wizard, and Voice Memo. In most cases, these applications draw objects of sizes: 320x320, 192x169, 96x80, and 64x34, respectively. The purpose of the second set of experiments is to validate the

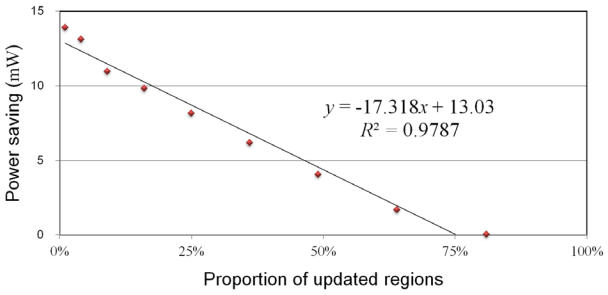


Figure 9: The Power Saving

effectiveness of *LPD* in the real-world environments on commercialized products. Note that *LPD* does not require any modifications in applications and *LPD* is applicable to any Tizen devices or X Window systems with Linux DRM and the `i80` display interface.

For each test case, we have conducted three experimental runs. Each experimental run consists of a continuous execution for 30 seconds. In order to get the average value of a continuous execution, an in-house power measurement device accumulates the energy consumed via the battery connection (supplied by V_{BATT}) and shows the average power over the 30 seconds.

The in-house power measurement device samples the current every 0.2 ms with the range of 0.6 mA to 4 A in the 0.01 mA granularity and with less than 1% of error. The measurement device sends the data to a tablet or a PC via a Bluetooth connection in real-time and allows the tablet or the PC to visualize or later analysis of the accumulated data. We have supplied 4.0 V to the device constantly in order to make the measurement and analysis simple.

Due to the technical difficulty, we have measured the whole power consumption of the device, not the power consumed by the display system only. The power consumed by other non-related devices such as GPU and network adapters is included. Thus, any visible power saving in the experiments is significant enough to motivate the adaption to commercial products; engineers invest huge effort and time to get additional minutes of battery life. If it is an extra hour for a 72-hour device, the responsible engineer may even be called a *hero* by his or her colleagues.

5.1 Synthetic Workload Benchmark Result

Table 1 and Figure 9 show the amount of power that *LPD* saves in the first set of experiments with the synthetic workload benchmark. As shown in the column

Table 1: Power-wise Synthetic Benchmark Result

Updated regions	Control (mW)	LPD (mW)	Power saving (mW)	Power saving (%)
320x320	209.03	208.99	0.04	0.02%
288x288	203.08	203.04	0.04	0.02%
256x256	196.84	195.18	1.66	0.84%
224x224	192.01	187.95	4.06	2.11%
192x192	187.83	181.64	6.20	3.30%
160x160	184.27	176.10	8.17	4.43%
128x128	181.43	171.60	9.84	5.42%
96x96	179.06	168.10	10.96	6.12%
64x64	177.35	164.25	13.10	7.39%
32x32	176.47	162.58	13.89	7.87%

of Updated regions of Table 1, we have executed the test application with various sizes of updated regions while fix the frame rate to 30 FPS. The results illustrate that the less data (= less updated region size) the display controller transfers, the more power saving we can get. In order to show the power-wise overhead of *LPD*, we have experimented with the full screen update that corresponds to the "320x320" row in Table1. In this test case, *LPD* cannot provide any benefit but incurs overheads only. However, the test results indicate that the power-wise overhead induced by *LPD* is ignorable. Surprisingly, *LPD* reduces 0.02% of power consumption. Because *LPD* adds a few CPU cycles per frame, we suspect that errors in power meters or variances in the experiments such as the temperature are responsible for this result.

Figure 9 shows power saving as a function of the update region size. We fit the observation point with a linear line in order to see if the amount of power saving is linearly related to the size of updated region as Eq. (4) suggests. The linear equation embedded in Figure 9 has the goodness-of-fit value of 0.98. Such a value implies that the model fits with the experimental results well.

The power saving of *LPD* appears to be more than expected if the proportion of updated regions is very small: the two left-most points in Figure 9. We speculate that the difference is due to the DVFS mechanism on the memory bus and memory interface. A DVFS mechanism for the memory bus and interface [9] can further save power by lowering the voltage and the frequency if the memory transmission is reduced. With the lower voltage and frequency, the energy consumption is no more linear to the memory bandwidth.

Based on the experiments, we can conclude that *LPD*

Table 2: Power Reduction of Real-World Applications

App	*LPD* power saving		Reduced traffic
	mW	%	kiB/s
W-launcher	0.22	0.20	-
Heart rate	0.39	0.58	134.6
Setup-wizard	1.14	1.52	7178.0
Voice memo	2.70	2.98	7165.5

is successful in saving energy when the proportion of updated region is small. Especially, if *LPD* is applied to smart phones or tablets equipped with higher resolution displays, the energy saving will be greater as suggested by this experiment and the model summarized in Eq. (4).

5.2 Experiments with Real-World Applications

Table 2 shows how much power *LPD* saves with actual applications running on Samsung Gear 2. In Table 2, the two sub-columns of "*LPD* power saving" show power saving in absolute values (mW) and in relative values (%). The column of "Reduced traffic" shows the memory bandwidth reduction. Table 2 suggests that *LPD* reduces power consumption of commercial applications running on a commercial product as well.

5.3 Overhead of *LPD*

LPD requires a few additional lines of codes in the middleware and kernel device drivers based on DRM. Therefore, *LPD* incurs additional overhead. We can infer the energy overhead of *LPD* by activating *LPD* for the cases where *LPD* is completely useless; i.e., the whole screen is updated every frame. For example, in Table 1, the case that "320x320" is updated represents such a case. As shown in Table 1, the energy overhead induced by *LPD* is -0.0462 mW. This result implies that the overhead of *LPD* is too scanty that the overhead is obscured by environmental variances. This is consistent with the amount of instructions added for the implementation of *LPD*; i.e., only several lines of trivial arithmetic instructions without loops or context switches are added to device drivers and middleware.

6 Future Work and Implications

LPD has been released with the X Window-based Tizen 2.3 commercial device, Samsung Gear S. However, in later versions, Tizen plans to use Wayland instead of the X Window System [23]. In order to keep the benefit of *LPD* for later Tizen versions, we will need to implement *LPD* on top of Wayland.

Further enhancement of *LPD* may draw out additional power conservation. That is, *LPD* may improve further by utilizing the characteristics of the DVFS-capable display bus such as MIPI-DSI. MIPI-DSI controller has various control modes: HSM (High Speed Mode), LPM (Low Power Mode), and ULPM (Ultra Low Power Mode). With a lot of display updates such as video playing, MIPI-DSI needs to operate at HSM, which supports bandwidth from 80 Mbps to 1 Gbps. Eliminating the transfer of unchanged regions, *LPD* may be able to reduce the bandwidth less than 80 Mbps. Then, MIPI-DSI can operate in the LPM mode, which consumes significantly less power than HSM.

LPD is an excellent example of vertical optimization that involved with several layers of the system. By allowing the kernel to accept simple yet performance critical hints that are readily available at middleware, we are able to use the given hardware more efficiently with minimal modifications and without any kernel hacks that deteriorates the maintainability of software. As an example of vertical optimization, *LPD* suggests that operating system architects should be well aware of the information that its upper layers have–the updated regions of the window system–and what its lower layers want–the `i80` display panel in *LPD*. *LPD* suggests that well-designed co-operation between multiple layers is extremely important.

LPD depends on the ability of the window system to recognize updated regions of the screen. The current implementation of the X Server depends on the correct operation of applications. That is, if an application declares that the whole screen is updated even though only parts of the screen are actually updated, and then *LPD* cannot save power. This implies that educating application developers for proper implementation or provision of a proper SDK tool is critical in deploying *LPD* and power saving. This indicates further need for vertical optimization going up through SDK, tools, and applications. Another aspect is that the UX design is extremely important in power saving; i.e., the updated region size of each frame matters significantly. We may conjecture that vertical power optimization should be extended even to UI/UX designs, which is already becoming important with the adoption of AMOLED displays; AMOLED consumes power differently depending on the colors and brightness of pixels.

7 Conclusion

LPD can lessen power consumption induced by memory operations and data transfers related with frame buffers. The first implementation of *LPD* has been applied to Samsung Gear 2 for the experimentation purpose. After confirming the stability and usability of *LPD*, we have successfully commercialized it for Gear S and released the complete source code for the public access. Even though we confined *LPD* to wearable Tizen devices only, contributing *LPD* to the mainline Tizen might be a trivial process. We are also ready to upstream *LPD* to the Linux kernel community and the infrastructural patch for *LPD* has been submitted and merged to the DRM tree for Linux 3.16. The main body of *LPD* is to be upstreamed to the Linux kernel community afterwards. Because *LPD* is not a compatibility breaking kernel hack, but a mainline upstream-able kernel feature, any Linux-based devices with the popular `i80` display interfaces can use *LPD* to save power.

The experimental results have shown that *LPD* saves significant amount of energy for wearable devices. If we save 5% of total energy for a device with 72 hours of life time, we extend additional 3 hours and 36 minutes of the life time. Besides, as discussed in Section 3, the energy saved by *LPD* might be larger for mobile devices with higher resolutions. More significantly, *LPD* does not require any modifications in hardware as long as the device has the de facto standard, `i80`. *LPD* does not incur noticeable overhead in CPU and *LPD* does not affect the visual quality of the display at all. Finally, *LPD* may be used with other display power saving mechanisms independently without any modifications in user applications.

8 Acknowledgments

We would like to thank Dr. Jong-Deok Choi, Dr. Hyogun Lee, and Dr. Sang-bum Suh for the support and advices. We would also like to express our special thanks to YoungJun Cho, a kernel graphics expert, who has been participated in the implementation and test of *LPD* for the commercialization of Samsung Gear series. We would like to show our gratitude to the other Tizen kernel and system framework developers for their commitment in the development of Tizen and its products. Comments from the anonymous reviewers, Dr. Chong-kwon Kim, and Geunsik Lim were extremely helpful in revising the paper.

9 Availability

LPD has been used for Samsung Gear S product running Tizen. Both userspace and kernel codes for Gear S, including *LPD*, are available for the public. You can access the kernel code for Gear 2 with *LPD* in the same site as well:

`http://opensource.samsung.com/`

If readers want to look at, understand, and contribute the *LPD*-related code, they may want to access the repositories of Tizen after creating an account at `http://tizen.org/`, which is opened to the public and operated by the Linux Foundation:

`http://review.tizen.org/`

References

[1] ANAND, B., THIRUGNANAM, K., SEBASTIAN, J., KANNAN, P. G., ANANDA, A. L., CHAN, M. C., AND BALAN, R. K. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 57–70.

[2] ARM LTD. Adaptive scalable texture compression. `http://www.arm.com/products/multimedia/mali-technologies/adaptive-scalable-texture-compression.php`. Accessed: 2015-01-30.

[3] ARM LTD. Arm frame buffer compression. `http://www.arm.com/products/multimedia/mali-technologies/arm-frame-buffer-compression.php`. Accessed: 2015-01-06.

[4] ARM LTD. Transaction elimination. `http://www.arm.com/products/multimedia/mali-technologies/transaction-elimination.php`. Accessed: 2015-01-06.

[5] CARROLL, A., AND HEISER, G. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 21–21.

[6] CHANG, N., CHOI, I., AND SHIM, H. Dls: Dynamic backlight luminance scaling of liquid crystal display. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 12*, 8 (August 2004), 837–846.

[7] CHENG, W.-C., HOU, Y., AND PEDRAM, M. Power minimization in a backlit tft-lcd display by concurrent brightness and contrast scaling. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (Washington, DC, USA, 2004), DATE '04, IEEE Computer Society, pp. 10252–.

[8] CHOI, I., SHIM, H., AND CHANG, N. Low-power color tft lcd display for hand-held embedded systems. In *Proceedings of Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)* (August 2002), pp. 112–117.

[9] CORBET, J. Better device power management for 3.2. *LWN* (Nov. 2011). `http://lwn.net/Articles/466230/`.

[10] CROXFORD, D., JONES, S., AND FLORDAL, O. Adaptive frame buffer compression, Nov. 2013. US Patent App. 13/898,510.

[11] GATTI, F., ACQUAVIVA, A., BENINI, L., AND RICCO', B. Low power control techniques for tft lcd displays. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (New York, NY, USA, 2002), CASES '02, ACM, pp. 218–224.

[12] HUNG, L., AND CHEN, C. Recent progress of molecular organic electroluminescent materials and devices. *Materials Science and Engineering: R: Reports 39*, 5 (2002), 143 – 222.

[13] JAPANESE SUPREME COURT. Precedent of a civil case of a consumer vs. panasonic. `http://www.courts.go.jp/app/files/hanrei_jp/686/080686_hanrei.pdf`, 2011. Accessed: 2015-01-06.

[14] KIM, H., CHA, H., AND HA, R. Dynamic refresh-rate scaling via frame buffer monitoring for power-aware lcd management. *Softw. Pract. Exper. 37*, 2 (Feb. 2007), 193–206.

[15] NYSTAD, J., LASSEN, A., POMIANOWSKI, A., ELLIS, S., AND OLSON, T. Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2012), EGGH-HPG'12, Eurographics Association, pp. 105–114.

[16] OTERHALS, J., CROXFORD, D., ERICSSON, L., NYSTAD, J., AND LILAND, E. Graphics processing systems, May 2011. US Patent App. 12/923,518.

[17] PANASONIC. 18650 cell scope / high power density trend. `http://www.slideshare.net/GIPC2011/20140207-panasonic-b2-b-gipc`, 2014. Accessed: 2015-01-27.

[18] PARK, J., CHUN, B., CHOI, Y., AND LEE, J. Method of reducing power consumption and display device for reducing power consumption, Aug. 14 2014. US Patent App. 13/928,334.

[19] PASRICHA, S., LUTHRA, M., MOHAPATRA, S., DUTT, N., AND VENKATASUBRAMANIAN, N. Dynamic backlight adaptation for low-power handheld devices. *IEEE Des. Test 21*, 5 (Sept. 2004), 398–405.

[20] SHIM, H., CHANG, N., AND PEDRAM, M. A compressed frame buffer to reduce display power consumption in mobile systems. In *Proceedings of Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC)* (January 2004), pp. 818–823.

[21] SINGHAR, A. Smart backlights to minimize display power consumption based on desktop configurations and user eye gaze, Apr. 1 2014. US Patent 8,687,840.

[22] THE MOVING PICTURE EXPERTS GROUP. Mpeg standards. `http://mpeg.chiariglione.org/standards`. Accessed: 2015-04-28.

[23] VANCUTSEM, J. Tizen ivi 3.0-m1 released. `https://lists.tizen.org/pipermail/ivi/2013-July/000563.html`, 2013. Accessed: 2015-01-19.

[24] WHELAN, R., AND GRINDSTAFF, M. Low power display refresh, Oct. 7 2004. US Patent App. 10/407,758.