Reducing Latency by Eliminating Synchrony

Min Hong Yun, Songtao He, and Lin Zhong
mhyun, songtaohe, lzhong@rice.edu
Rice University, Houston, TX

Abstract

Drawing or dragging an object on a mobile device is annoying today because the latency is manifested spatially with an obvious gap between the touch point and the line head or dragged object. This work identifies the multiple synchronization points in the input to display path of modern mobile systems as a major source of latency, contributing about 30 ms to the overall latency.

We present Presto, an asynchronous design of the input to display path. By focusing on the main application and relaxing conventional requirements of no frame drop and no tearing effects, Presto is able to eliminate much of the latency due to synchrony. By carefully guarding against consecutive frame drops and limiting the risk of tearing to a small region around the touch point, Presto is able to reduce their visual impact to barely noticeable. Using a prototype based on Android 5, we are able to quantify the effectiveness, overhead and user experience of Presto through both objective measurements and subjective user assessment. We show that Presto is able to reduce the latency of legacy Android applications by close to half; and more importantly, we show this reduction is orthogonal to that by other popular approaches. When combined with touch prediction, Presto is able to reduce the touch latency below 10 ms, a remarkable achievement without any hardware support.

1. INTRODUCTION

User-perceived latency is the delay from when a user acts on a system to when the system's response is externalized on the display. Short latency is critical to a good user experience [5]. Because human users are unable to perceive latency of many tens of ms between visual cause and effect, latency of several tens of ms has been considered adequate for point/selection-based interaction [38]. On today's smartphones and tablets, the latency is over 60 ms according to the literature [9] and our own measurement. Unfortunately, the use of touchscreen manifest latency into a spatial gap between the touch point and the visual effect, i.e. the line head of drawing or the object being dragged [41]; a latency of 60 ms produces an obvious, annoying gap. Figure 1 illustrates this.

The goal of this work is to reduce the latency of touchscreen interaction on modern mobile systems for unmodified applications, both native and web. Our key insight is that the input to display path within modern mobile systems are synchronized at multiple points, with a periodical signal that coincides with the display refreshing, as show in Figure 2. Such synchronization contributes significantly to the overall latency. Our key idea is to eliminate the synchrony in this path. Modern mobile systems, however, employ the synchrony for three important reasons: a consistent frame rate, no frame drop, and absolutely no tearing effect. In §2, we analyze the input to display path, highlight the three points of synchronization, and revisit their reasons.

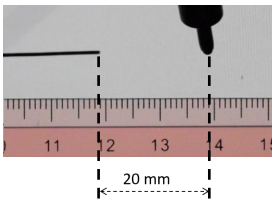

Figure 1: Touchscreen drawing translates latency (82 ms Android) into a visible gap as the pen moves at 25 cm per second and the line head falls behind (Autodesk Sketch, camera captured).

In §3, we present Presto, an asynchronous design of the input to display path, for mobile systems. The key rationale behind Presto is that the three reasons for synchrony must change with today's hardware and software as elaborated. Presto exploits the liberation from the above three reasons by applying two novel designs to the main application under interaction. *Just-in-time trigger*, or JITT, eliminates synchronization between the input subsystem and the application and that inside the output subsystem. It triggers the input subsystem to deliver events to the application so that the latter's output will be ready for display right before the next display refresh. *Position-aware rendering*, or PAR, alleviates the latency from the synchronization point in the display subsystem by selectively allowing the application to directly write into the graphics buffer that is being externalized to the display.

We report an Android 5-based implementation of Presto and evaluate its effectiveness in latency reduction, overhead, and assurance in user experience with both objective measurements and subjective assessment. Our measurements show that Presto reduces the latency by 32 ms on average for legacy applications, with a power overhead that can be eliminated with emerging SDK support. The effectiveness is obvious as presented in §5. Importantly, we show that the latency reduction resulting from Presto is orthogonal to that from known techniques such as touch prediction used by iOS 9. Double-blind user evaluation of legacy applications with and without Presto clearly demonstrates that Presto improves the user experience with touchscreen interaction without noticeable side effects. The prototype implementation is described in §4 and the evaluation is presented in §5.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052557

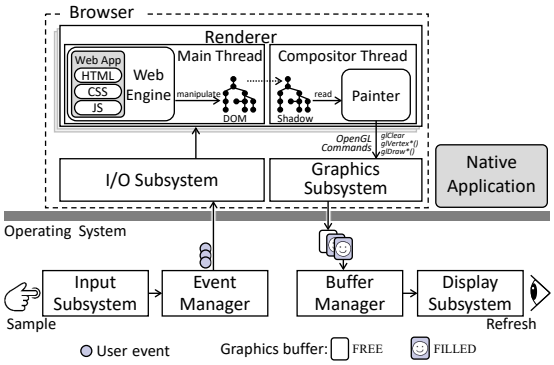

Figure 2: From events to the display: the event manager batches events from the input subsystem and delivers them to the application; the application then draws a frame on a buffer supplied by the buffer manager. The buffer manager transfers buffer ownership to the display subsystem. A browser adopts three-part model and imposes additional layer on the place of the native application.

Although we evaluated Presto with touchscreen interaction, it will reduce latency for other forms of interactions. Importantly, for augmented reality applications with a head-mounted display, the head movement can also manifest latency into spatial displacement of virtual objects. By reducing the latency due to synchronization, Presto is likely to be effective there as well.

On the other hand, Presto is limited in two important ways. First, it only deals with latency resulting from synchronization. The majority of the rest of the latency comes from the input hardware. Therefore, techniques such as faster input hardware and touch prediction can complement Presto. We show that Presto, when combined with touch prediction of 30 ms, can reduce the overall latency below 10 ms. Second, although Presto is a system solution that supports all applications, it should not be blindly applied to all applications because of the power overhead and risk of tearing from PAR. Instead, we anticipate the developer and the user to decide whether Presto should be enabled for a specific application. These are elaborated in §7.

2. UNDERSTANDING LATENCY

In this section, we present an in-depth analysis of the input to display path on modern mobile systems. Our analysis pinpoints to a fundamental source of latency: when a user input event propagates through a mobile system, many of its subsystems have to wait for a global synchronization signal, instead of processing the event immediately. Modern mobile systems opt for this synchronized design in order to avoid tearing and drop frames and to keep a consistent frame rate, all at the cost of latency.

2.1 From an event to the display

Based on an understanding of mainstream mobile OSes, i.e., iOS and Android, we abstract in Figure 2 the process in which input events are processed and eventually result in a screen update. The process includes five software subsystems: input, event manager, application, buffer manager, and display. All except application are part of the OS (not necessarily in kernel space though).

The *input* subsystem includes the input device driver. It samples the physical world and produces software events.

The *event manager* is per-application. It buffers events from the input and delivers them to the application. The buffering is necessary because the input subsystem produces the events faster than the display refreshes. High-rate events are necessary because of application's desire for smooth visual effects.

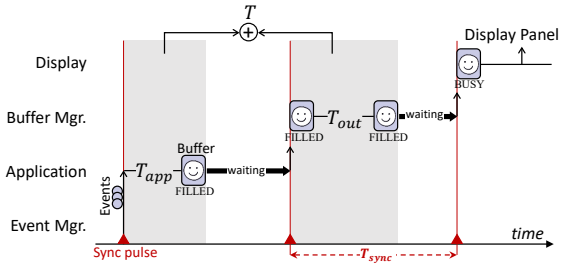

Figure 3: The timeline to process events and display a frame: the sync pulse fired by the display controller triggers the event and buffer managers, causing waiting and delay.

The *buffer manager* is also per-application. It manages the application's *graphics buffers*. The application processes the input events, takes a FREE buffer, marks it BUSY_{app}, draws a frame on it and then marks it FILLED.

The *display* subsystem includes the software part of the composer. It takes FILLED buffers from multiple applications, marks them as BUSY_{disp}, and handles them to the hardware, which composes the buffers and sends the composition to the display panel serially. After that, the display subsystem marks the buffers as FREE. Because composing is done by specialized hardware, it adds negligible latency. The display controller refreshes the display panel and fires a sync pulse periodically, with the period of T_{sync}. In modern mobile systems, T_{sync} is typically $1/60$ s [15, 40].

Web applications: Browsers themselves are native applications; modern browsers are platforms that run web applications implemented in HTML, CSS, and JavaScript. Common to many modern browsers [39, 31, 42] are three parts as depicted in Figure 2: I/O subsystem, renderer, and graphics subsystem. On mobile systems, these parts are implemented as separate processes (Chromium and Safari with WebKit2) or threads (Firefox). The renderer is per-application for security reasons and can access I/O and GPU only indirectly via the I/O subsystem and the graphics subsystem, respectively. The renderer has two threads: main and compositor threads. The main thread runs the web engine (e.g., WebKit or Blink); the engine loads a web application, creates and updates its document object model (DOM) tree. The compositor thread commands the graphics subsystem based on the DOM tree. Because the DOM tree is shared between the two threads, the browser avoids locking by keeping a shadow tree in the compositor thread. When the web engine cannot complete updating (e.g., inserting new nodes or changing attributes of nodes) the DOM tree before the next screen update, it commits partial updates to the shadow tree so that it can complete pending updates while the compositor thread is issuing drawing commands. The graphics subsystem, which has privileged access to the GPU, draws on a graphics buffer using the received commands.

2.2 Synchrony introduces latency

At the cost of long latency, a modern mobile OS guarantees three visual goals: a consistent frame rate, no frame drops, and no tearing effects. To achieve the goals, a mobile OS introduces synchrony to its design in temporal and spacial manners. In time, the event and buffer managers strictly synchronize with the sync pulse produced by the display controller; in space, it assumes a buffer can not be read and written at the same time. Figure 3 provides a timeline from an application to the display panel in modern mobile OSes.

Synchrony in time: In the legacy design, both the event and buffer managers synchronize with a periodical signal, a.k.a. the sync pulse, fired by the display subsystem when refreshing. The event manager waits for a sync pulse to deliver buffered events

to the application. This buffering introduces an average latency of $0.5 \cdot T_{sync}$. Assume the application takes T_{app} to process the events, produce a frame and write it into a graphics buffer. It will wait another $(T_{sync} - T_{app})$ until the next sync pulse so that the buffer manager can process the buffer[1]. Android reduces this by triggering the event manager 7.5 ms after the sync pulse [16]. In this case, the latency would be $(T_{sync} - T_{app} - 7.5_{ms})$.

The buffer manager waits for a sync pulse to change graphics buffers' ownership among the application, display subsystem and itself. Assuming this process takes T_{out}, this synchronization introduces an average latency of $(T_{sync} - T_{out})$ because the buffers will be externalized only at the next sync pulse. These synchronizations together ensure a *consistent frame rate*. Synchronization of the buffer manager additionally ensures *no frame drops*. No matter how quickly an application finishes drawing, the buffer manager transfers buffer ownership only on a sync pulse.

Synchrony in space: Noticeably, the buffer manager does not give a BUSY_{disp} buffer to the application, avoiding the same buffer being read by the display and written by an application at the same time. This synchronous buffer access is sufficient but not necessary to *avoid tearing effects*. However, the buffer manager does not have better strategy because it has no idea about which pixels have been changed from one frame to the next, i.e., dirty region. This strategy makes an average latency of $0.5 \cdot T_{sync}$ due to the display refreshing necessary because the application has to finish writing in a buffer before the display starts to externalize it. As a result, any BUSY_{disp} buffer has to wait for the next display refreshing to be sequentially externalized, introducing an average latency of $0.5 \cdot T_{sync}$.

All together, we estimate the average latency due to the synchrony as

$$3 \cdot T_{sync} - (T_{app} + T_{out}) \quad (1)$$

For a typical Android application, this latency is about 34.9 ms with $T_{sync} = 1/60$ s and the 7.5 ms optimization deducted. This accounts for close to half of the latency we observe on Android devices. One naïve way to reduce this latency is to simply reduce T_{sync}. This would not only increase power consumption systemwide, but also require more expensive hardware. Our approach, in contrast, aims at eliminating the synchrony without increasing T_{sync} or requiring new hardware.

2.3 Overall latency

Considering the average latency of the input hardware (T_{touch}), we can derive the average end-to-end latency below using a similar analysis and the assumption that $T_{app} \leq T_{sync}$:

$$T_{touch} + 3 \cdot T_{sync} \quad (2)$$

of which $3 \cdot T_{sync}$ is from when the event manager receives events to when the display finishes rendering the resulting frame (Figure 3).

If we completely eliminate the latency due to the synchrony, the best average latency will be

$$(2) - (1) = T_{touch} + T_{app} + T_{out} \quad (3)$$

When the input subsystem produces events faster than the application can consume, the above best latency is unachievable because the event manager must buffer events. With existing hardware, our solution, Presto, adds $0.5 \cdot T_{sync}$ of event buffering latency to the best latency. New display systems where the display sync pulse is configurable and adjustable at runtime, e.g., G-Sync [34], may help reduce this even buffering latency.

[1]This analysis assumes $T_{app} \leq T_{sync}$. If $N \cdot T_{sync} < T_{app} \leq (N+1) \cdot T_{sync}$, the added latency would be $(N+1) \cdot T_{sync} - T_{app}$.

Additional latency for web applications: The above analysis applies to all native applications, including web browsers. Web applications, however, are subject to another layer of indirection and more latency. First, a web application experiences overhead due to scripting, leading to longer T_{app} than that of native applications. The overhead includes running JavaScript and updating the DOM tree based on its result. Modern JavaScript engines compile JavaScript code to expedite its execution; however, an input event has to go through multiple layers to reach the compiled code: from the main thread in the renderer to the web engine to the DOM tree to the JavaScript engine. Likewise, the engine's execution results also have to go through multiple layers in the reverse path.

Second, the browser increases the latency when subscribing a wrong sync pulse, due to Android's 7.5 ms optimization (§2.2). The browser synchronized with the sync pulse to produce a frame before the next screen update. However, the 7.5 ms-delayed sync pulse makes the browser produce a frame 7.5 ms after the sync pulse and miss the next screen update. This always adds a latency of $(T_{sync} - 7.5_{ms})$. Presto to be presented in §3 does not cause this issue by excluding this optimization in its design.

Finally, we must note that IPC overhead among the three parts of a browser is negligible. The I/O subsystem sends the sync pulse and input events via a Unix socket to the renderer, which sends drawing commands to the graphics subsystem, via shared memory. Both contribute negligible latency according to our measurements on Nexus 6.

3. DESIGN OF PRESTO

We next present the design of Presto. Compared to today's systems, Presto almost halves the latency by judiciously relaxing the visual constraints. In particular, Presto eliminates the synchrony in the legacy design with two key techniques, *just-in-time trigger*, or JITT, and *just-enough pixels*, or JEP. JITT eliminates the synchronization of the event and buffer managers. It aims to get as many input events to the app as the resulting frame will be ready by the next display refresh. The JITT buffer manager transfers the buffer ownership to the display subsystem immediately after the app finishes drawing, without waiting for a sync pulse. JEP and its approximation, *position-aware rendering*, or PAR, further alleviate the atomic use of buffers by judiciously allowing an app to write into a BUSY_{disp} buffer that is being externalized by the display.

Visual Constraints Revisited: The key insight behind Presto is that the three visual constraints, i.e., consistent frame rate, no frame drop, and no tearing effect, can be judiciously traded for reducing latency with better user experience. First, HCI research has shown that many interactions on mobile devices require latency far lower than what modern mobile devices can deliver. Ng. et al [32, 33] showed that the just-noticeable difference (JND) latency for object dragging on the touchscreen is $2-11$ ms. Microsoft went further to argue for 1 ms latency for touchscreen interactions [41].

Second, the three visual goals met by the legacy design are not absolute. For some applications and interactions, they are not necessary at all, especially on modern mobile hardware and software. Hardware improvements, i.e., faster CPU, GPU and larger memory, have enabled a consistent frame rate of 60 fps on modern mobile systems. Recent studies [6, 7] have shown that users cannot perceive changes in frame rate when it is above 30 fps. Similarly, frame drops can be allowed if they are not consecutive and the frame rate is kept above 30 fps. Importantly, drawing on touchscreen usually has visual effects limited to the touched position. Tearing effects would be barely noticeable by human eyes or even high-speed cameras. Indeed, they are almost indistinguishable from the effect of latency as highlighted by Figure 4.

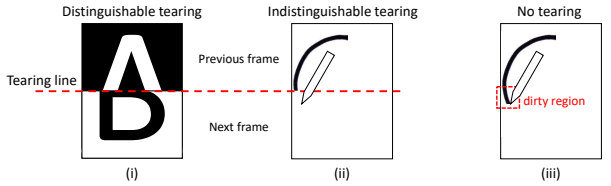

Figure 4: Tearing effect may happen when the display switches from one frame to the next in the middle of externalizing the first. As a result, the screen will show the early part of the first frame and the late part of the second, joined at the tearing line. If the tearing line cuts a large dirty region, the tearing effect can be visible and annoying as is in (i). If the dirty region is small, like in the cases of drawing, the tearing effect is indistinguishable from the effect from latency, as is in (ii), when compared to the perfect case in (iii).

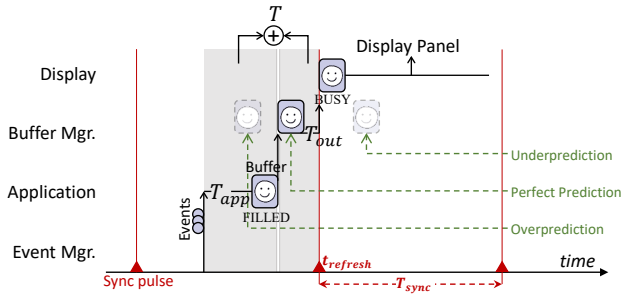

Figure 5: JITT removes synchronization in the event and buffer managers. It decides when the event manager delivers events to the application so that the buffer manager would deliver the buffer filled by the application right before the next display refresh. To do so, it must predict how long it will take from the event delivery to the buffer delivery, or T. Overprediction ($T' > T$) leads to an increase in latency by ($T' - T$); underprediction ($T' < T$) leads to an increase in latency by T_{sync}.

Finally, it can be profitable for user experience to trade these visual goals for shorter latency. Janzen and Teather [22] showed that latency affects user performance with touchscreen interaction more than frame rate does. Presto also carefully drops delayed frames in order to cut overall latency.

3.1 JITT: Just-In-Time Trigger

JITT removes synchronization in the event and buffer managers. With JITT, the event manager judiciously decides when to deliver buffered events to the app; and the buffer manager transfers buffer ownership as soon as the application finishes drawing, without waiting for the sync pulse. Ideally, the buffer manager would deliver the buffer filled by the application's response right before the next display refresh. Recall that we denote the time it takes the application to process the events and fill the buffer as T_{app}, the time it takes the buffer manager to transfer the buffer ownership to the display subsystem as T_{out}. For brevity, we denote $(T_{app} + T_{out})$ as T.

In the ideal case with JITT, no events would have to wait more than $(T + T_{sync})$ for their application response to externalize, with average being $(T + 0.5 \cdot T_{sync})$. This is illustrated by the perfect prediction in Figure 5. Therefore, knowing when the display refreshes next, denoted by $t_{refresh}$, JITT must predict T, and let the event manager deliver the events at $(t_{refresh} - T')$ where $'$ indicates prediction.

T_{app} and T_{out} can be easily predicted using history. Much of the prediction algorithm is system-specific and we will revisit when reporting the implementation (§4.1). Below, we focus on one important design issue. Inaccurate prediction increases latency of JITT. An overprediction $(T' > T)$ makes the event manager deliver events too soon. That is, if events arrive between $(t_{refresh} - T')$ and $(t_{refresh} - T)$, the corresponding frame would wait for the screen refresh and increase the average latency by $(T' - T)$. This is illustrated by the overprediction in Figure 5. An underprediction $(T' < T)$ makes the event manager wait too long to deliver the buffered events and as a result, the buffer manager will not be able to transfer the resulting graphics buffer to the display subsystem by the next display refresh, adding an entire T_{sync} to the average latency. This is illustrated by the underprediction in Figure 5. Apparently, the latency penalty is significantly higher in the case of underprediction.

JITT copes with underprediction in two ways. First, it favors overprediction between overprediction and underprediction. That is, it looks for the upper end when using history. Moreover, with prediction T', instead of triggering the event manager at $(t_{refresh} - T')$, JITT calculates when the last event would arrive before $(t_{refresh} - T')$ and triggers the event manager when this event arrives. This trick essentially adds a variable offset to T' in favor of overprediction. Second, JITT recovers from underprediction by dropping the frame in the buffer delayed due to underprediction. Importantly, this recovery mechanism does not drop two frames in a row. When underprediction happens, the buffer manager will have two FILLED buffers when JITT triggers it: one delayed and the other newly produced. Then, the buffer manager drops the older buffer by marking it as FREE and transfers the newer one to the display subsystem. If JITT underpredicts one more T in a row, the buffer manager does not drop the delayed frame anymore but propagates the delay until no underprediction happens or the application stops producing frames. The worst case is when T_{app} changes abruptly and the JITT buffer manager drops every other frame, the frame rate becomes half, or 30 fps on modern mobile systems.

3.2 JEP: Just-Enough Pixels

As explained in §2.1, in modern mobile systems, when an application requests a graphics buffer, the buffer manager will give it a FREE one. Therefore, the application cannot write into the BUSY_{disp} buffer that is being externalized by the display subsystem. This atomic buffer access avoids tearing but adds a latency of $0.5 \cdot T_{sync}$ on average as discussed in §2.2. JEP reduces this latency by judiciously allowing the application to write into the BUSY_{disp} buffer, without tearing.

JEP leverages partial-drawing APIs like [24, 17] and a modern mobile display trend: an in-display memory from which the display panel reads pixels, not directly receiving from the composer [21, 37]. The key idea is to make the atomic area smaller, i.e., the dirty region of the new frame, and let the display subsystem take only the dirty region to compose and update the in-display memory only before the display panel starts externalizing the dirty region. This is possible without tearing because a modern display externalizes a frame sequentially, pixel by pixel and updating only the dirty region reduces the memory copy between the buffer and the display subsystem, e.g., by 7178.0 KB/s [21].

Specifically, JEP needs to answer two questions: (1) where is the starting point of the dirty region? That is, in how many pixels will the display externalize before reaching the dirty region? (2) how fast is the display subsystem externalizing pixels? The use of partial-drawing APIs answers (1). The answer to (2) is independent

of applications and can be accurately profiled. For example, in our prototype, we find the display subsystem externalizes 221 M pixels per second.

Because most legacy mobile applications do not use the partial-drawing APIs and not all mobile displays feature the internal memory, we next present PAR, an approximation of JEP, to support legacy applications and displays.

3.2.1 PAR: Position-Aware Rendering

To support legacy applications and displays, PAR allows an application to write in the $\texttt{BUSY}_{disp}$ buffer that is being externalized by the display subsystem. To minimize the risk of tearing effects caused by concurrent buffer accesses, PAR must be confident that the application would finish writing into the buffer BEFORE the display subsystem starts externalizing a dirty region. Therefore, in addition to the previous two questions to JEP, PAR must answer a third question: how long will it take the application to finish drawing into the buffer? Notably, the answer is essentially T_{app}; its prediction is already available from JITT as described in §3.1. Like in JITT, underprediction is more harmful than overprediction in PAR: underprediction risks tearing effects while overprediction only decreases latency reduction.

To further limit tearing effects, we exploit the fact that many applications will have visual effects and henceforth dirty regions limited to around the touched position; and tearing in this area is barely distinguishable from effect of latency as shown in Figure 4. Presto will apply PAR only if the dirty region is within a predefined rectangle, 200 by 200 pixels in our implementation, centered at the latest touch point. This also simplifies the implementation. Presto will first check if there is any change outside the rectangle around the touch point, i.e., any dirty region outside it. If so, it stops. Otherwise, PAR estimates if the application can finish writing before the display reaches the edge of the rectangle. If yes, it will respond to the application with the $\texttt{BUSY}_{disp}$ buffer.

To check if there is a dirty region outside the rectangle, PAR can leverage help from the application, the answer to (1). For legacy applications that do not use the partial-drawing APIs, PAR compares the two adjacent frames by sampling. We discuss how we implement it and its overhead in §4.2 and §5.2.4, respectively.

4. IMPLEMENTATION

We next describe our prototype implementation of Presto using Android 5 (Lollipop). The implementation includes about 1125 SLOC in the input and output subsystems of Android.

In Android 5, the event manager includes a library `libinput`, and a sync pulse receiver `DisplayEventReceiver` in Android runtime library `libandroid_runtime`. Both are in an application's address space. In the `libinput` library, the `InputConsumer` object receives events from the input subsystem through a Unix socket and buffers them. `InputConsumer` delivers the buffered events to the application when `DisplayEventReceiver` receives a sync pulse from the display subsystem.

The buffer manager is `BufferQueue`, which is part of Android GUI library `libgui` and allocates buffers and manages their ownership. Note that we use `BufferQueue` to refer to three classes: `Core`, `Producer`, and `Consumer`. `BufferQueue` is indirectly synchronized with the sync pulse by responding to requests from the display subsystem. Each application window has its dedicated buffer manager (in `SurfaceFlinger`'s address space for performance reasons). As a result, `BufferQueue`s from different applications are independent from each other.

The display subsystem includes `SurfaceFlinger`, which receives FILLED buffers from multiple applications' `BufferQueue`s and sets up the hardware composer. `SurfaceFlinger` also relays the sync pulse from the hardware composer to the event manager.

4.1 JITT: Just-in-Time Trigger

We implement JITT by revising the event manager (`libinput` and `DisplayEventReceiver`) and buffer manager (`BufferQueue`). The predictor for T_{app} tracks T_{app} history and predicts based on a simple algorithm that averages the recent 32 measurements of T_{app}, or roughly half a second. We empirically set T_{out} to 3.5 ms based on profiling of `BufferQueue` and `SurfaceFlinger`. The constant time is conservatively determined to give `SurfaceFlinger` enough time to transfer the ownership of multiple applications' graphics buffers, from `BufferQueue` to the hardware composer.

To trigger the event manager, we modified `DisplayEventReceiver` to intercept the sync pulse from the display subsystem and re-fire it at the predicted time $(t_{refresh} - T')$. When `BufferQueue` is requested to give a FILLED buffer by the `SurfaceFligner`, it waits until the predicted time $(t_{refresh} - T_{out})$ and then responds with the latest FILLED buffer just before the next screen refresh.

4.2 PAR: Position-Aware Buffer Manager

We implement PAR by modifying `BufferQueue` and Android's ION memory manager. Recall that when the application requests a buffer, PAR responds with the $\texttt{BUSY}_{disp}$ buffer in the application's buffer manager only if it is confident that the application would finish writing into the buffer BEFORE the display subsystem starts externalizing a dirty region. Our implementation conveniently obtains the prediction of how long it will take the application to finish writing into the buffer from JITT, i.e., T'_{app}. We profile that the display subsystem reads the $\texttt{BUSY}_{disp}$ buffer at 221 M pixels per second.

If the application does not already provide information about the dirty region, e.g. via an SDK like [21], our implementation identifies the starting point of the dirty region by modifying Android ION's `ioctl()` syscall to compare frames in software. We compare the frames in the kernel space because graphics buffers are not directly accessible from the user space for security reasons. `BufferQueue` passes a buffer's ION `fd` to the kernel via the syscall. Then, the kernel finds the corresponding memory area represented in `scatterlist` [4], samples 1% of the frame, and then compares them with those of the previous frame. One can increase the number of samples to track dirty region more accurately; however, 1% from a 2560×1440 screen (Nexus 6) is sufficient to check the dirty regions of applications updating the entire screen, such as animation and scroll.

5. EVALUATION

Using the prototype implementation, we aim at answering the following questions regarding Presto.

- How effective is Presto in reducing latency? how much does each of its two key techniques contribute?
- Is its effectiveness orthogonal to that of other popular techniques, namely event prediction [3, 26, 27, 35, 40]?
- What tradeoffs does Presto make, in terms of power consumption and the visual goals dear to the legacy design?
- How do end users evaluate the overall performance of Presto?

5.1 Evaluation Setup

We evaluate our implementation on Google Nexus 6 smartphones with Android 5.0 (Lollipop) and Linux kernel 3.10.40. The smartphone has a 5.96″ 2560 × 1440 AMOLED display, 2.7 GHz quad-core CPU, and 600 MHz GPU. During the evaluation, we use a

DotPen stylus pen with a tip of $1.9\,\mathrm{mm}$ [10], instead of finger, to find out the touched position with high accuracy.

5.1.1 Latency Measurement

We measure the interaction latency with two methods. The first one is *indirect*, by combining calibration, analysis, and OS-based time logging. It is applicable to all applications. The second is *direct* based on camera capture and video analysis. It is, however, only applicable to applications whose visual effects are amenable to our video analysis. We use the indirect method to report latencies for legacy benchmarks; we use the direct method to provide in-depth insight along with the in-house benchmark.

Indirect Measurement: The indirect measurement method breaks down the end-to-end latency into three parts and deal with each differently: (1) from physical touch to the touch device driver, (2) from the touch device driver to the display subsystem, and (3) from the display subsystem to display externalization.

We measure the latency of (1) by using a microcontroller and two light sensors (API PDB-C142, response time: $50\,\mathrm{us}$): the microcontroller continuously polls the sensor output at $1\,\mathrm{KHz}$. The first light sensor besides the screen shoots a laser beam from the other side. When the stylus pen crosses the laser beam, the microcontroller detects the change of the light sensor output and logs a timestamp. When the touch device driver receives an event crossing the beam, it turns on the built-in LED, which takes $1.5\,\mathrm{ms}$. The second light sensor, placed above the LED, detects this so that the microcontroller logs the second timestamp. We estimate the latency of (1) as the difference between these timestamps: $28.0\pm1\,\mathrm{ms}$.

We measure the latency of (2) by logging two timestamps in software: when the touch device driver receives an event and when the ownership of the resulting buffer is transferred to the display subsystem. Notably this latency is where Presto makes a difference.

We estimate the latency of (3) based the y-coordinate of the touch event logged in software as described above. Since the display panel illuminates pixels sequentially top-down after a sync pulse, we estimate when the pixels of the touched area illuminate as $T_{sync} \cdot y/H$ where H is the screen height measured in pixel number.

Direct Measurement: For the in-house benchmark, we are able to measure the user-perceived latency by analyzing video record. What a camera can precisely capture are the locations: that of the square (L_s) in response to a touch and that of the pen (L_p) in each frame. Therefore, we estimate the velocity of the pen movement (v) from its locations in consecutive frames. By calculating how long it would take the pen to travel from the touched location (L_s) to the current pen location (L_p), we obtain the latency as $(L_s - L_p)/v$. This estimation, however, relies on the assumption that the velocity of the pen does not change abruptly from frame to frame. Due to the high frame rate, i.e., $60\,\mathrm{Hz}$, this assumption is largely true and also confirmed by our own measurement.

A camera also introduces errors due to its frame rate. We use a Nikon D5300 camera with 60 Hz frame rate and $1/500\,\mathrm{sec}$ shutter speed. The frame rate would introduce a random latency uniformly distributed between 0 to $16.7\,\mathrm{ms}$ (T_{sync}). Therefore, we deduce this random variable when reporting the latency measurement.

We compare the latency derived from the indirect measurement of the in-house application against with its direct measurement with stock Android, Presto (JITT) and Presto (JITT+PAR). The both measurements are within $2.5\,\mathrm{ms}$ from each other. The difference is smaller than their standard deviation and more importantly, one order of magnitude smaller than the latency reduction by Presto.

5.1.2 Benchmarks

We evaluate Presto with both legacy applications and an in-house application. The legacy applications include ten native drawing applications: the top five each from the Drawing & Handwriting and Calligraphy categories of the Google Play Store on Jan 26, 2016. Not surprisingly, we could not find many web applications for Android Chromium that support drawing: the latency is too long for good user experience. In the end, we found three and included all for evaluation. For these applications, we measure the latency using the indirect method. The five from Drawing & Handwriting are Notepad+ Free (N+), Autodesk Sketch (AD), Handrite Note (HN), Bamboo Paper (BP), and MetaMoJi Note Lite (MM). The five from Callligraphy are Calligraphy HD (CY), Calligrapher (CR), INKredible (IK), Brush Pen (BP), and HandWrite Pro Note (HP). The three web applications are DeviantArt Muro (MR), Literally Canvas (LC), and Zwibbler (ZB). We note that our benchmarks exclude applications that update a frame without a user event, e.g., games. This is because the indirect measurement method must associate a frame update with a user event.

We also employ several in-house native and web applications that have an identical design. The native ones are implemented for both Android and iOS using OpenGL ES 2.0. The web application uses HTML5 `canvas` element. All implementations draw a 115×115 square and a horizontal line on a touched position. As the pen moves, it drags the square and line along. These applications are valuable for three reasons. (*i*) They allow us to understand the accuracy of the indirect measurement of legacy applications as reported in [44]. (*ii*) They allow us to compare our Android-based Presto prototype with iPad Pro with Apple Pencil, a cutting-edge touch device commercially available, using the same OpenGL ES code base. *(iii)* Because the application has bare minimum functionality for touch interaction, it allows us to better understand the power overhead of Presto.

5.1.3 Interaction and Trace Collection

Short of a programmable robotic arm, we try our best to produce repeatable traces of interaction with the benchmarks. For each benchmark, we interact by manually moving the pen repeatedly from one end of the screen to the other vertically in portrait orientation, with a steady speed for 150 seconds. Post collection analysis shows an average speed of $68\,\mathrm{mm}$ per second, with a standard deviation of 12. All traces are available at [1].

5.2 Latency Reduction by Presto

We next answer the three questions about the latency reduction by Presto: how much is it? how is it related to that from other techniques? what does Presto trade for it? The measurement shows that Presto with JITT only and with (JITT+PAR) reduces the average latency of our benchmarks from $72.7\,\mathrm{ms}$ to $54.4\,\mathrm{ms}$ and $41.0\,\mathrm{ms}$, respectively. This reduction eliminates all latency from synchronization. Moreover, as we anticipated, the reduction from Presto is orthogonal from that of another important technique, touch prediction, employed by iPad Pro. When combined with touch prediction of $30\,\mathrm{ms}$, Presto is able to reduce the latency of our in-house application below $10\,\mathrm{ms}$.

5.2.1 Presto *reduces latency by* $32\,\mathrm{ms}$

Figure 6 shows how much each of the two techniques reduces the latencies of legacy applications. On average, Presto reduces the latency by $32\,\mathrm{ms}$. To appreciate the significance of this reduction, we note that Deber et al [8] showed that even a small latency reduction, i.e. $8.3\,\mathrm{ms}$, brings a perceptible effect in touchscreen interactions. Latency reduction for some applications, e.g., Autodesk(AD), is larger than average latency caused by the synchrony, i.e., $34.9\,\mathrm{ms}$ (§2.2). This is because when an application occasionally fails to

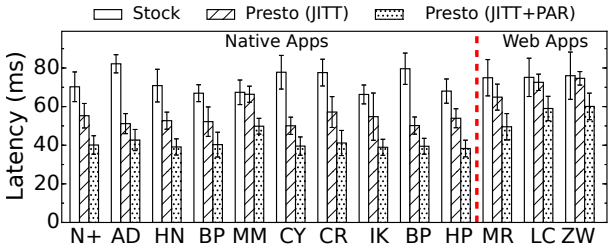

Figure 6: Presto consistently reduces the latency of the legacy benchmarks, by 32 ms on average.

finish drawing by the next display refresh, JITT drops this delayed frame while stock Android keeps it and propagates the delay to all subsequent frames. The frame drop by Presto is not perceptible to users as we will see in§5.3.

Notably, different benchmarks see different reductions in latency from Presto. Presto is most effective for those that have large latency to begin with, i.e., Autodesk (AD), Calligraphy (CY), Calligrapher (CR), and Brush (BP). Presto is the least effective for MetaMoji (MM), reducing the latency by 17.6 ms only. Our analysis reveals that this is because its average T_{app} is the longest among all benchmarks. As a result, it has the shortest latency due to synchronization and gives Presto the least opportunity.

Generally, the latency reductions on the web applications are smaller than native applications because the browser's web engine takes longer T_{app}. Specifically, the latency reductions on Literally Canvas (LC) and Zwibbler (ZB) are smaller than DeviantArt Muro (MR) because their web pages are heavier.

5.2.2 Presto *beats iPad Pro*

Using our in-house native application, we are able to compare Presto on Nexus 6 with iOS on iPad Pro, the state-of-the-art touch device widely in use. iPad Pro employs two techniques to reduce the latency. First, it doubles the input sampling rate for Apple Pencil [2], from 120 Hz to 240 Hz, to reduce latency. Second, the iOS SDK provides predicted events for the next frame (16 ms), a technique called *touch prediction* [40], which hides the latency. Importantly, both techniques reduce the latency impact by T_{touch} in Equation 2, which makes Presto complementary. The doubled sampling rate reduces it by 8 ms, and the touch prediction further reduces it by 16 ms as shown in the left column of Figure 7. Because neither technique is available on Android, we measure the in-house application on iPad Pro with four configurations as reported in the left column of Figure 7: normal stylus pen without touch prediction, Apple Pencil without touch prediction, normal stylus pen with touch prediction, and Apple Pencil with touch prediction. The results clearly show that both the faster input sampling rate and touch prediction help reduce the latency for iPad Pro, with the best latency being 42.9 ms. Impressively, Presto is able to reduce Android's latency to even lower, 33.0 ms, even without fast input sampling or touch prediction.

For the in-house web application, Presto reduces its latency to 55.5 ms, below all iPad Pro configurations except Apple Pencil with Touch Prediction. This is because the web application has a long latency to begin with.

5.2.3 Presto *brings orthogonal benefits*

In principle, the effectiveness of Presto is orthogonal to that of faster input and touch prediction because Presto eliminates latency resulting from synchronization, i.e., Equation 1, and the latter primarily reduce latency resulting from the input hardware, i.e., T_{touch} in Equation 2. With our in-house application, we implement touch prediction that predicts into the future from 0 to 32 ms. Presto reduces the latency by eliminating the synchronization points. Figure 8 shows how Presto and touch prediction complementarily reduces the latency. The leftmost group in the figure does not have predicted events, i.e., touch prediction of 0 ms. Clearly, for touch prediction of various time, Presto demonstrates almost the same effectiveness in latency reduction. Interestingly, Presto with touch prediction of about 30 ms is able to reduce the average latency below 10 ms, a rather remarkable achievement by a software-only solution.

5.2.4 *Tradeoffs by* Presto

Presto trades off other computing goals for short latency: it judiciously allows frame drops and tearing, and may incur power overhead through PAR. When we try out the benchmarks with Presto, we could not see any effects usually associated with frame drops or tearing. Our double-blind user study, reported in §5.3, confirms this independently. Below we report objective data regarding frame drops, tearing risk, and power overhead.

By design, Presto guarantees no consecutive frame drops. In the worst case, it would drop 50 % of the frames (every other frame). Our measurement, reported in Figure 9, shows a much lower rate for our benchmarks, with the worst case being 8 % (Bamboo (BP)).

There is no direct way we could observe the occurrences of tearing: as shown in Figure 4, even if tearing happens and is captured by camera, it would be extremely hard to tell it from the effect of latency. Instead, we measure how frequent underprediction of T_{app} happens. As shown in §3.2.1, an underprediction of T_{app} is a necessary but not sufficient condition for tearing to happen. Therefore, the frequency of underprediction can be considered as an upper bound for that of tearing. Figure 9 shows the frequencies of underprediction for the legacy benchmarks. HandWrite (HP) has the highest frequency of underprediction (17 %). Bamboo (BP) has the highest frequency (13 %) amongst the five benchmarks used in the user study. These frequencies are at most suggestive of how often tearing may happen. None of the authors could see any effects due to tearing; nor did the participants in our user study.

We use a Monsoon Power Monitor [30] to measure the power consumption of Presto in Nexus 6. We disable all wireless communications and dim the LCD backlight to the minimum level. We measure the power consumption of the in-house application during 60 seconds of touchscreen drawing for each of the following configurations: without Presto, with Presto (JITT), Presto (JITT+PAR without frame comparison), and Presto (JITT+PAR). Their power consumption and standard deviations are 2017 ± 120, 2075 ± 115, 2024 ± 110, and 2564 ± 201 mW, respectively.

We would like to highlight two points regarding the power overhead. First, JITT increases the power consumption only slightly, well below the standard deviation. PAR (without frame comparison) decreases the power consumption to be barely indistinguishable from that of the stock Nexus 6, i.e. 2024 ± 110 vs. 2017 ± 120. This is because PAR reduces activities of the buffer manager. Second, the frame comparison needed for PAR contributes most of the power overhead, an 27% increase. Because in our measurement we disabled all wireless interfaces and dimmed the LCD backlight to minimum, the percentage increase for real-world usage will be much lower. More importantly, using frame comparison to determine the dirty region is not practically necessary because the GPU and application already have the information. Some SDKs, e.g.,[14], already make this information available via an API, e.g., `invalidate(Rect dirty)`. With such APIs, this overhead would be eliminated.

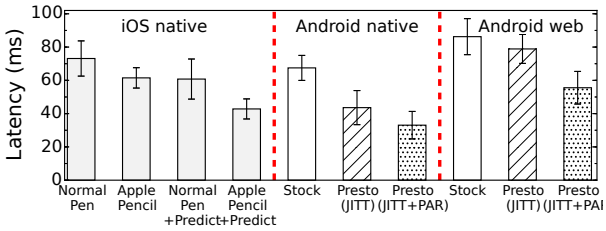

Figure 7: Latency of our in-house app: iOS native, Android native, and web on Android. Presto reduces the latency of Android native and web versions below that of iOS native even with Apple Pencil (and touch prediction for Android native).

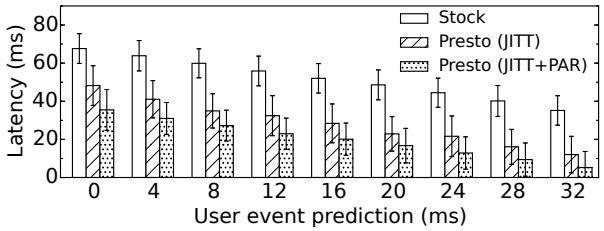

Figure 8: Latency of Presto plus touch prediction for our in-house application: the effectiveness of Presto is complementary to that of touch prediction. X axis is the time into the future predicted.

5.3 User Evaluation

When we try the benchmarks with Presto, it is visually obvious that Presto reduces latency significantly. None of the authors are able to notice any tearing effects or frame drops. Nevertheless, In defense against any possible experimenter's bias, we perform a double-blind user study to evaluate Presto subjectively.

We recruited 11 participants via campus-wide flyers. They were students and staff members from various science and engineering departments, between 19 and 40 years old, with three women. All had at least two-month experience with an Android device with a display bigger than 5.5 inches.

Each participant came to the lab by appointment and was given two Nexus 6 smartphones that are identical except one has stock Android, the other Presto. The smartphones are marked A and B, respectively. Neither the participant nor the study administrator knew which one is stock. The participant was then asked to use their finger or a stylus pen to try out the five top Android applications from Drawing & Handwriting. They were allowed to try as long as they wished; and all finished in 10 to 45 minutes. After each application, the participant answered three questions: (*i*) which device is faster: A, B or same? (*ii*) if you chose A or B, to what extent do you agree with the statement that the latency difference is obvious? (1 to 5 with 1 being *strongly disagree* and 5 being *strongly agree*) (*iii*) other than the latency, describe any difference you observe. For post-mortem analysis, we recorded the hand-smartphone interaction of all except two participants with a GoPro Hero 4 camera at 240 Hz. The recordings are available from [1].

5.3.1 Findings: quantitative and qualitative

Figure 10 presents participants' answers to the first question. Not surprisingly, more than half of the participants consider Presto to be faster in each of the benchmarks. For Autodesk (AD), 10 out of 11 participants considers Presto is faster. This corroborates the measurement presented in Figure 6, which shows Audtodesk (AD) sees the largest latency reduction amongst the five. To our surprise and puzzlement, a same participant reported the stock Android is faster in Notepad+ (N+) and Bamboo (BP). We checked the video record, and it was obvious to us that Presto was clearly faster in both the applications. One theory to explain this is that the participant mistook A with B when answering the question. Nevertheless, we are wary that the same theory can be used to argue the participant's responses for the other three applications were also mistaken. Overall, the data suggests that participants overwhelmingly felt that Presto is faster. For those who considered Presto to be faster, the average of their responses to the second question is 3.5, indicating the latency difference is obvious to them.

Our participants were asked if they observe any difference beyond latency. None reported any effects that may result from inconsistent frame rate, frame drop, or tearing, such as application's fluctuating response time, screen flickering, and screen overlap. Indeed most of their comments are about secondary effects due to latency difference. Two participants did notice some details about how Presto actually works. One remarked about MetaMoji (MM) that Presto "seems to catch up quicker than" the stock Android. The other observed similar effects with Autodesk (AD) but worded it differently: the stock Android has "smooth curves;" Presto is "not as soft as" the stock Android. By that, the participant was referring to the same effect that when drawing a line, the line with Presto sometimes jumps to the touch point, or "catch up quicker" in the words of the first participant.

Sample size: 11 participants are not many. The question is: is our conclusion that Presto is faster statistically significant? We use dependent t-test for paired samples to analyze the statistical significance of the answers to the first question shown in Figure 10. The dependent t-test [45] is used to calculate the probability (p) of sampling error when the same subjects are exposed to both samples (applications on stock Android and Presto). For each benchmark, we count how many people considered that on Presto is faster and calculated p using the dependent t-test. Since $p < 0.01$, the difference is statistically significant.

6. RELATED WORK

Latency has long been recognized as a key to user experience. There is a rich body of literature that goes back to more than half a century ago. In addition to those that have already been discussed, we discuss five groups of recent works as related to Presto. To our best knowledge, Presto would be the first in the public domain that identifies synchrony in the operating system design as a major source of latency and eliminates it.

Resource Management: A faster computer reduces the application execution time $(T_{app} + T_{out})$ (§2.2). The authors of [12, 23, 11, 13, 43] favor interactive applications in OS resource management to reduce their execution time. Many others, e.g., [19, 20, 18], leverage cloud or cloudlet to improve the interactive performance of mobile applications. Because these solutions do not reduce the latency due to the synchrony, they are complementary to Presto: they reduce latency when $(T_{app} + T_{out}) > T_{sync}$ while Presto is most effective when $(T_{app} + T_{out}) < T_{sync}$. Additionally, when $(T_{app} + T_{out}) < T_{sync}$, these solutions improve the opportunity for Presto by reducing $T_{app} + T_{out}$ as in Equation 1.

Speculation: Event prediction and speculative execution have also been studied to conceal latency. Event prediction, or touch prediction in Apple's term, is widely used for virtual reality with the head mounded display. To compensate for prediction errors, researchers have explored speculative execution [28] and post image processing [29]. All these solutions, as discussed in §5.2.3, are complementary to Presto.

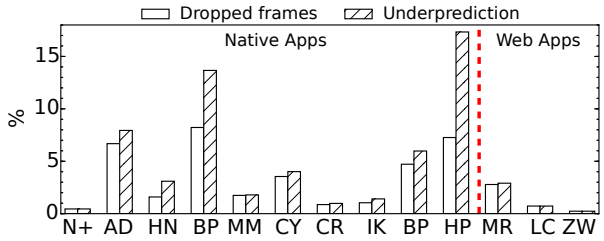

Figure 9: Presto occasionally experiences frame drops and underprediction.

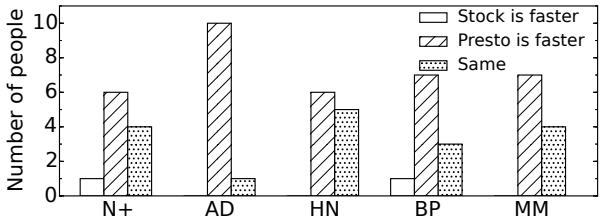

Figure 10: Number of participants answering Question (i) in each of the three ways: which device is faster: A, B or same?

Specialized Hardware: As part of a testbed for studying touch latency, Ng et al. report an ultra-low latency touch system [33, 32] that achieves a latency as short as 1 ms. The system employs a proprietary touch sensor with a very high sampling rate (1 KHz), FPGA-based low-latency (0.1 ms) data processing, and an ultra-high speed digital light projector (32 000 fps). With completely custom software and hardware, it is not feasible for mobile systems, let alone supporting any legacy applications as Presto does.

Alternatives to VSync: Games on non-mobile devices often provide an asynchronous, or *vsync-off*, mode to reduce latency. In the vsync-off mode, the event manager delivers input events to the game whenever the latter is ready; otherwise, the manager buffers the events. Similarly, the buffer manager changes graphics buffers' ownership without waiting for a sync pulse, even when the display is reading. This vsync-off mode, unfortunately, can introduce tearing effects anywhere on the screen [36] because it blindly ignores the sync pulses. JITT avoids this problem by changing graphics buffers' ownership only when a sync pulse is fired; PAR checks dirty regions and confines the tearing effects, if any, to a small area under the touch position.

NVIDIA's G-Sync [34] reduces latency in a way very similar to JITT but requires proprietary GPU and display. JITT times the event manager carefully so that the resulting frame will be ready to display right before the next sync pulse. In contrast, a G-Sync GPU generates a sync pulse when it finishes rendering to synchronize the event and buffer managers, and the display. On the other hand, PAR and G-Sync are complementary. With G-Sync, PAR can calculate backwards when the display should refresh to reach a dirty region immediately after the region is rendered. When combined with G-Sync, Presto can control the display refresh and reduce the event manager's buffering latency from $0.5 \cdot T_{sync}$ (§2.3) to $0.5 \cdot (T_{app} + T_{out})$ on average.

7. CONCLUDING REMARKS

In this work, we identify synchrony in modern mobile systems as a major source of latency. We present Presto, an asynchronous design for user interaction. By focusing on the main application and relaxing conventional requirements of no frame drop and no tearing effects, Presto is able to eliminate much of the latency from synchrony. By carefully guarding against consecutive frame drops and limiting the risk of tearing to a small region around the touch point, Presto is able to reduce their visual impact to barely noticeable. Using a prototype realization, we show that Presto is able to reduce the latency of legacy Android applications by close to half; and more importantly, we show this reduction is orthogonal to other popular approaches. When combined with touch prediction, Presto is able to reduce the touch latency below 10 ms, a remarkable achievement without any hardware support.

Below we offer some thoughts about future directions and how Presto may be adopted/deployed.

Latency due to Input Hardware: Presto is able to reduce the average latency from about 70 ms to 40 ms. Where does the rest of latency come from? Our investigation has pointed to the inside of the input subsystem, which contributes about 30 ms in Android systems. This includes the hardware time for scanning capacitance changes on the touch sensor, converting analog signals to digital, and communicating to the CPU [25]. This latency can be reduced in two ways. First, exemplified by Apple Pencil, is to increase the input sampling rate, whose effectiveness is shown in Figure 7. The more effective way, however, is touch prediction, as exemplified by iOS 9, whose effectiveness is shown in Figure 7 and Figure 8.

A Reflection on Best Practice: While synchrony is a major source of latency for all applications and Presto works for unmodified legacy applications, it is *not a panacea*. This is particularly true for PAR, which risks tearing effects in a small region around the touch point. While it so happens none of the benchmarks used in your evaluation would manifest tearing to human eyes, it is also easy for the authors to imagine an application that will, e.g., one that displays the coordinate of touch next to the touch point.

Therefore, instead of applying Presto blindly to all applications, the application developer and the end user should make the call. Presto can perfectly co-exist with the traditional design and be applied to applications selectively. Indeed, it is perfectly fine to enable Presto only for certain features of an application or turn it on and off at runtime [44].

The slight power overhead of PAR, due to the dirty region determination by frame comparison, is another factor that the developer and end user should consider. Concerned with this overhead, the developer should either disable it or disclose the dirty region information using APIs supported by SDK like [14]. The end user or the operating system on behalf of them should decide if such overhead is acceptable based on user preference and energy availability.

ACKNOWLEDGEMENTS

This work was supported in part by NSF Award CNS #1422312. The authors thank Pu Dong and Abeer Javed who administrated the double-blind user study.

8. REFERENCES

[1] RECG download page. `http://download.recg.org`.

[2] Apple. Apple pencil. `http://www.apple.com/apple-pencil`.

[3] T. Asano, E. Sharlin, Y. Kitamura, K. Takashima, and F. Kishino. Predictive interaction using the delphian desktop. In *Proc. ACM UIST*, 2005.

[4] J. E. Bottomley. Dynamic dma mapping using the generic device. `https://www.kernel.org/doc/Documentation/DMA-API.txt`.

[5] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, and B. Shneiderman. Determining causes and severity of

end-user frustration. *International Journal of Human-Computer Interaction*, 2004.

[6] K. T. Claypool and M. Claypool. On frame rate and player performance in first person shooter games. *Multimedia Systems*, 2007.

[7] M. Claypool, K. Claypool, and F. Damaa. The effects of frame rate and resolution on users playing first person shooter games. In *Electronic Imaging*. Int. Society for Optics and Photonics, 2006.

[8] J. Deber, R. Jota, C. Forlines, and D. Wigdor. How much faster is fast enough?: User perception of latency & latency improvements in direct and indirect touch. In *Proc. ACM CHI*, 2015.

[9] D. E. Dilger. Agawi TouchMark contrasts iPad's fast screen response to laggy Android tablets. `http://appleinsider.com/articles/13/10/08/agawi-touchmark-contrasts-ipads-fast-screen-response-to-laggy-android-tablets`, 2013.

[10] Dot-Tec. Dot pen. `http://dot-tec.com`.

[11] Y. Endo and M. Seltzer. Improving interactive performance using TIPME. In *Proc. ACM SIGMETRICS*, 2000.

[12] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer. Using latency to evaluate interactive system performance. In *Proc. USENIX OSDI*, 1996.

[13] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on a commodity OS. In *Proc. USENIX OSDI*, 2002.

[14] Google. GLSurfaceView. `http://developer.android.com/reference/android/opengl/GLSurfaceView.html`.

[15] Google. Graphics architecture. `http://source.android.com/devices/graphics/architecture.html`.

[16] Google. Implementing graphics. `http://source.android.com/devices/graphics/implement.html`.

[17] Google. invalidate(). `https://developer.android.com/reference/android/view/View.html`.

[18] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through Flip-Flop replication. In *Proc. ACM MobiSys*, 2015.

[19] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code offload by migrating execution transparently. In *Proc. USENIX OSDI*, 2012.

[20] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proc. ACM MobiSys*, 2014.

[21] M. Ham, I. Dae, and C. Choi. LPD: Low power display mechanism for mobile and wearable devices. In *Proc. USENIX ATC*, 2015.

[22] B. F. Janzen and R. J. Teather. Is 60 fps better than 30?: The impact of frame rate and latency on moving target selection. In *Proc. ACM CHI*, 2014.

[23] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proc. ACM SOSP*, 1997.

[24] Khronos Group. glScissor(). `https://www.khronos.org/opengles/sdk/docs/man/xhtml/glScissor.xml`.

[25] S. Kolokowsky and T. Davis. *Not All Touchscreens are Created Equal - How to ensure you are developing a world class touch product*. Planet Analog: `http://www.cypress.com/file/98261`, 2010.

[26] E. Lank, Y.-C. N. Cheng, and J. Ruiz. Endpoint prediction using motion kinematics. In *Proc. ACM CHI*, 2007.

[27] J. J. LaViola. Double exponential smoothing: An alternative to Kalman filter-based predictive tracking. In *Proc. Eurographics Wrkshp. Virtual Environments*, 2003.

[28] K. Lee, D. Chu, E. Cuervo, Y. Degtyarev, S. Grizan, J. Kopf, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proc. ACM MobiSys*, 2015.

[29] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. In *Proc. SIGGRAPH Symp. Interactive 3D Graphics and Games (I3D)*, 1997.

[30] Monsoon. Monsoon power monitor. `https://www.msoon.com`.

[31] Mozilla. Off main thread compositing. `https://wiki.mozilla.org/Platform/GFX/OffMainThreadCompositing`, 2015.

[32] A. Ng, M. Annett, P. Dietz, A. Gupta, and W. F. Bischof. In the blink of an eye: Investigating latency perception during stylus interaction. In *Proc. ACM CHI*, 2014.

[33] A. Ng, J. Lepinski, D. Wigdor, S. Sanders, and P. Dietz. Designing for low-latency direct-touch input. In *Proc. ACM UIST*, 2012.

[34] NVIDIA. G-sync. `http://www.geforce.com/hardware/technology/g-sync`, 2014.

[35] P. T. Pasqual and J. O. Wobbrock. Mouse pointing endpoint prediction using kinematic template matching. In *Proc. ACM CHI*, 2014.

[36] T. Petersen. GPU boost 3 and SLI. `https://www.technopat.net/sosyal/konu/video-what-is-nvidia-fast-sync.329258`, 2016.

[37] E. Petillon. Demystify DSI I/F: `http://www.ti.com/lit/an/swpa225/swpa225.pdf`. Texas Instruments, 2012.

[38] S. C. Seow. *Designing and engineering time: The psychology of time perception in software*, chapter 3. Addison-Wesley Professional, 2008.

[39] The Chromium Projects. Multi-process architecture. `https://www.chromium.org/developers/design-documents/multi-process-architecture`, 2008.

[40] P. Tsoi and J. Xiao. Advanced touch input on iOS: Increasing responsiveness by reducing latency. The Apple Worldwide Developers Conference `https://developer.apple.com/videos/play/wwdc2015/233`, 2015.

[41] C. Velazco. Microsoft envisions a future with super-fast touchscreens. `http://techcrunch.com/2012/03/09/microsoft-demos-super-fast-touchscreen-but-will-they-ever-make-it-to-market`, 2012.

[42] WebKit. Webkit2 - high level document. `https://trac.webkit.org/wiki/WebKit2`, 2016.

[43] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proc. USENIX OSDI*, 2008.

[44] M. H. Yun, S. He, and L. Zhong. Polypath: Supporting multiple tradeoffs for interaction latency. *arXiv preprint arXiv:1608.05654*, 2016.

[45] D. W. Zimmerman. A note on interpretation of the paired-samples t test. *Journal of Educational and Behavioral Statistics*, 22(3):349–360, 1997.