POLYPATH: Supporting Multiple Tradeoffs for Interaction Latency

Technical Report 2016-08
Min Hong Yun, Songtao He, and Lin Zhong
Rice University, Houston, TX

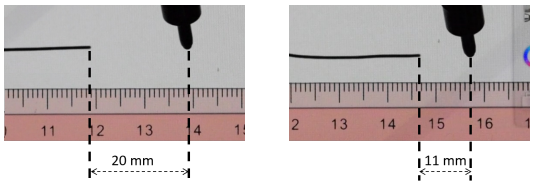
Abstract

Modern mobile systems use a single input-to-display path to serve all applications. In meeting the visual goals of all applications, the path has a latency inadequate for many important interactions. To accommodate the different latency requirements and visual constraints by different interactions, we present POLYPATH, a system design in which application developers (and users) can choose from multiple path designs for their application at any time. Because a POLYPATH system asks for two or more path designs, we present a novel fast path design, called Presto. Presto reduces latency by judiciously allowing frame drops and tearing.

We report an Android 5-based prototype of POLYPATH with two path designs: Android legacy and Presto. Using this prototype, we quantify the effectiveness, overhead, and user experience of POLYPATH, especially Presto, through both objective measurements and subjective user assessment. We show that Presto reduces the latency of legacy touchscreen drawing applications by almost half; and more importantly, this reduction is orthogonal to that of other popular approaches and is achieved without any user-noticeable negative visual effect. 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

The input-to-display path, or *I2D path* for short, is an important operating system (OS) service because it determines the user-perceived latency of interaction. Today's mobile OSes employ the same path design for all interactions, a system design we call MONOPATH. This path design therefore must meet the visual goals of all applications: consistent frame rate, no frame drops, and no tearing effects. It achieves this with a coarse-grained design at the cost of long latency, over 60 ms [11, 30, 57]. Although this latency may be fine for point/selection-based interactions [54], it is annoying for others. For example, touchscreen-based drawing and dragging manifest latency as a spatial gap between the touch point and the visual effect [59]; a latency of 60 ms produces an obvious, annoying gap, as illustrated in Figure 1. The same is true for augmented-reality interactions based on head-mounted displays. The original motivation of our work is to reduce the latency for such demanding interactions.



(a) Stock Android (b) Presto

Figure 1: Touchscreen drawing translates latency, e.g., 82 ms, into a visible gap, e.g., 20 mm, as the pen moves and the line head falls behind (Autodesk Sketch, camera captured).

We quickly realized that latency reduction is not free. Without specialized, expensive hardware like that used in [44], one has to make tradeoffs between the visual goals and latency. Luckily, we find these goals are not necessary or can be relaxed for many interactions under modern hardware and software. All these point to the MONOPATH design of modern mobile OSes as the fundamental problem. To cater to the different latency requirements and visual constraints of diverse interactions, we argue that mobile OSes should follow POLYPATH and support multiple I2D path designs in the same system.

Our guiding principles for POLYPATH are twofold: (*i*) different applications and different parts of an application can employ different path designs; and (*ii*) application developers and users should decide which part of an application employs which path design.

In this paper, we report our design of POLYPATH that supports unmodified legacy applications. The key insight behind the design is that the interface between an application and the rest of the I2D path is clean and independent; each group of events delivered by the input to the application can be handled by a given path design, independently of the group before or after. Our POLYPATH system provides an asynchronous API for developers and users to bind a path design to an application; it further ensures that an input event only experiences one path design, a property we call *path integrity*.

Because a POLYPATH system asks for two or more path designs, we provide a novel I2D path design, called Presto. Compared to the path used in today's MONOPATH systems, Presto almost halves the latency

by judiciously allowing frame drops and tearing, making a very different tradeoff. In particular, Presto overcomes the coarse granularities of the legacy path design through two key techniques. *Just-in-time trigger* eliminates strict synchronization in the path with the display. *Just enough pixels* allows the I2D path to operate on only updated pixels, or dirty regions, of a frame.

We report an Android 5-based implementation of POLYPATH that supports two path designs: Android legacy and Presto. We evaluate its effectiveness in latency reduction, overhead, and user experience with both objective measurements and subjective assessment. Our measurements show that Presto reduces the latency by $32\,\mathrm{ms}$ on average for top drawing applications from Android Play Store, with a power overhead that can be eliminated with SDK support. The effectiveness is obvious from Figure 1. 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. When combined with touch prediction, Presto is able to reduce the touch latency below $10\,\mathrm{ms}$, a remarkable achievement without any hardware support. Double-blind user evaluation demonstrates that for the drawing applications tested, Presto improves the user experience without noticeable side effects.

In summary, we make the following contributions:

- We present a general model for the I2D path and identify that coarse granularity in today's I2D path design contributes significantly to the interaction latency. We show that this legacy design sacrifices latency to strictly meet several visual goals that are not always necessary today.
- We provide a design and implementation of POLYPATH that supports multiple I2D path designs in the same system. POLYPATH allows application developers and users to make different tradeoffs for latency without affecting other applications.
- As part of our POLYPATH system, we present Presto, a novel I2D path design that reduces latency by almost half. We provide a prototype implementation of Presto based on Android 5 that supports unmodified legacy applications and evaluate it with both objective measurements and subjective assessment.

2 I2D Path and Its Tradeoffs

In this section, we present a conceptual model for the I2D path to understand its design tradeoffs. We show that the I2D path design of today's MONOPATH systems represents a specific tradeoff point in a large design space between latency and other computing goals. By showing many other possible, desirable tradeoff points, we motivate the need for POLYPATH operating systems in which multiple I2D path designs are supported.

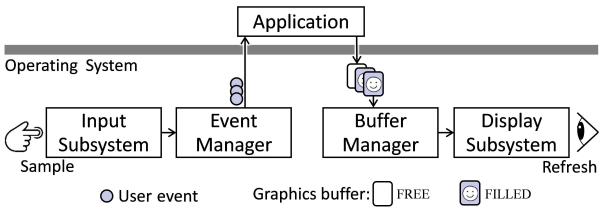


Figure 2: A general model for the I2D path: the event manager batches events from the input subsystem and delivers them to the application; the application then draws a frame on a buffer supplied by the buffer manager. The buffer manager transfers buffer ownership to the display subsystem.

2.1 I2D Path Model

Based on an understanding of mainstream mobile OSes, i.e., iOS and Android, we devise a five-part model for the I2D path as shown in Figure 2. The model includes 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 sampling rate is typically $120\,\mathrm{Hz}$ [58] but can range from 80 to $240\,\mathrm{Hz}$.

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.

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 $1/60\,\mathrm{s}$ [20, 58].

2.2 Coarse-Grained I2D Path Design

At the cost of long latency, the I2D path in today's mobile OSes guarantees three visual goals: a consistent frame rate, no frame drops, and no tearing effects. It achieves it with a design that is coarse-grained in both time and space. In time, its event and buffer managers strictly synchronize with the sync pulse produced by the display controller; in space, it assumes a buffer can not

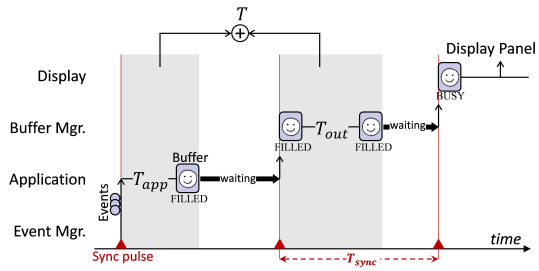


Figure 3: The timeline for the legacy I2D path in action: the sync pulse fired by the display controller triggers the event and buffer managers, causing waiting and delay.

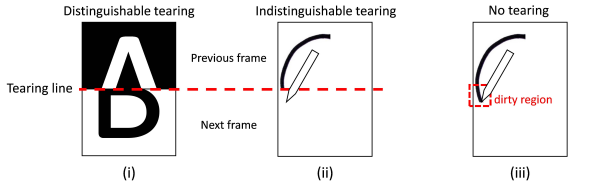


Figure 4: Tearing effect may happen when the display switches from one frame to the next in the middle of externalizing the first. As a result, the screen will show the early part of the first frame and the late part of the second, joined at the tearing line. If the tearing line cuts across a large dirty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

be read and written at the same time. Figure 3 provides a timeline for the legacy I2D path.

Synchronization: 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. 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. Therefore, synchronization at the event manager introduces an average latency of $(T_{sync} - T_{app})$. Android reduces this by triggering the event manager $7.5\,\mathrm{ms}$ after the sync pulse [21]. 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.

Atomic Buffer: 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 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 coarse-granularities as

$$2.5 \cdot T_{sync} - (T_{app} + T_{out}) \quad (1)$$

For a typical Android application, this latency is about $26.6\,\mathrm{ms}$ with $T_{sync} = 1/60\,\mathrm{s}$ and the $7.5\,\mathrm{ms}$ 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}. However, this would incur proportionally higher power consumption by the display subsystem. Even worse, when reducing T_{sync}, the subsystems will have to process proportionally faster because their processing time must be masked by T_{sync}, leading to system-wide proportionally higher power consumption and the use of expensive hardware.

2.3 Design Tradeoffs in I2D Path Design

The legacy path design analyzed above represents one particular point of tradeoff between latency and other computing goals. In some sense, it represents an extreme point: it efficiently and strictly meets all three visual goals, at the cost of latency. For many important latency-sensitive interactions, these goals can be relaxed, especially in view of the recent hardware and software development. As a result, *different tradeoffs are possible*, especially in favor of short latency.

First, HCI research has shown that different interactions have different latency requirements [7, 8]. It is known that the human-perceptible threshold for causality is about $100\,\mathrm{ms}$ [40, 54]. That is, if we click a button on screen and the button changes within $100\,\mathrm{ms}$, we would barely notice the delay. As a result, the $100\,\mathrm{ms}$ latency has been considered as adequate for keyboard/mouse-based interactions as well as touchscreen-based point/selection. In contrast, Ng. et al [43, 44] showed that the just-noticeable difference (JND) latency for object dragging on the touchscreen is $2-11\,\mathrm{ms}$, much shorter than

what modern mobile systems can deliver. Microsoft went further to argue for 1 ms latency for touchscreen interactions [59]. Latency reduction, however, is not free. Microsoft achieves it 1 ms latency only with expensive, specialized hardware. Given the hardware, latency reduction often requires more computation, e.g, event prediction [35] and speculation [37], or relaxing the visual goals, as explained below.

Second, the three visual goals met by the legacy I2D path 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, 9] 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. For another example, 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.

Moreover, it can be profitable for user experience to trade these visual goals for shorter latency. Janzen and Teather [29] showed that latency affects user performance with touchscreen interaction more than frame rate does. Our fast path design, Presto, to be presented in §4, also carefully drops delayed frames in order to cut overall latency. We believe that it should be up to the application developers and users to determine what tradeoffs are profitable.

Finally, on battery-powered mobile devices, the visual goals may be traded for lower power consumption. For example, on Nexus 6, lowering the frame rate from 60 to 30 fps reduces the overall system power consumption by 300 mW, or 20%, when running Angrybird.

In summary, hardware and software advancements described above make more tradeoffs between latency and other computing goals possible. Because the latency of today's MONOPATH system is inadequate for many interactions, we argue for a POLYPATH system design in which multiple path designs making different tradeoffs for latency coexist. In a POLYPATH system, application developers and users can decide when to apply which path design to which application.

3 Design of POLYPATH

Because all existing mobile OSes are MONOPATH, we face many important decisions when designing POLYPATH. In this section, we elaborate these decisions.

By introducing multiple path designs, POLYPATH first faces the problem of naming and binding. That is, in a POLYPATH system, an application must be able to *name* a path design and be able to use it by *binding* to it. Indeed, much of our POLYPATH design involves answering questions about the naming and binding. In this section, we speak of binding in an abstract way because different systems may implement it differently.

First, we must support legacy applications that are designed with the MONOPATH system in mind. This is possible via two design choices. (*i*) First, as apparent from Figure 2, there are two interfaces between an application and the rest of path: event delivery from the event manager and the buffer exchange with the buffer manager. As long as these two interfaces are kept unchanged, a path design should support unmodified legacy applications. (*ii*) Moreover, the naming of path and the binding between path and application must be achieved outside the application. In our POLYPATH design, they are realized by an OS module, called *path manager*. The path manager records the path preference of each application during app installation and updates the record via a system API it exports. The API has the simple form of `ApplyPath(app_name, path_name)`. Importantly, the decision to support legacy applications further allows us to focus on the OS part of I2D path, which includes the event and buffer managers. As a result, when we say *path design* in this work, we are referring to the design of the path, event and buffer managers.

Second, we must decide on the lifetime of the binding between an application and a path design. That is, when is a binding created and when does it change? In one extreme, the path manager can decide binding for each sequence of events delivered to the application and as a result, can change the binding from one sequence of events to the next. We consider this fine granularity as unnecessary. Instead, we opt for bind-by-need strategy similar to call-by-need evaluation [26]. That is, once a binding is created, at the time of the application launch, it lasts until the application or the system explicitly asks for a change, via the path changing API. This design invokes the path manager much less frequent and therefore has higher efficiency and better reliability.

Finally, when and how can the path binding of an application be changed? We note that the invocation of `ApplyPath()` can be asynchronous, i.e., it can happen any time during the application's lifetime. Notably, the path design consists of two disjoint parts, linked by the application: the *event manager* and the *buffer manager*. If the path design is changed after the event manager delivers the events to the application but before the buffer manager gives the application a buffer, the events will experience the event and buffer managers from two different designs. This behaviors violates the expectation of developers and users that an event should experience

the same path design, a property we call *path integrity*. Therefore, at the invocation of `ApplyPath()`, if the event manager has already buffered events, the path manager ensures that these events follows the current path; otherwise, it binds the new path to the application for the incoming events.

We note that this POLYPATH system design naturally supports any number of paths to be added or removed from the system. When the system cannot find the path required by an application, it can fall back to a default. Additionally, applications using the same path design in POLYPATH are as isolated from each other as they would be in the MONOPATH system.

4 Presto: A Fast Path Design

Because POLYPATH asks for two or more I2D path designs and existing mobile OSes only provides one, we next present a novel I2D path design, called Presto. Compared to the path used in today's MONOPATH system, Presto almost halves the latency by making a different tradeoff between latency and other computing goals. In particular, Presto eliminates the coarse granularities of the legacy path design, as discussed in §2.2, through two key techniques, *just-in-time trigger*, or JITT, and *just-enough pixels*, or JEP. Speaking of tradeoff, Presto judiciously allows frame drops (JITT) and tearing (JEP) in favor of short latency.

JITT eliminates the synchronization of the event and buffer managers. It aims to get as many input events to the application 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 application 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 application to write into a BUSY_{disp} buffer that is being externalized by the display.

4.1 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 application; 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

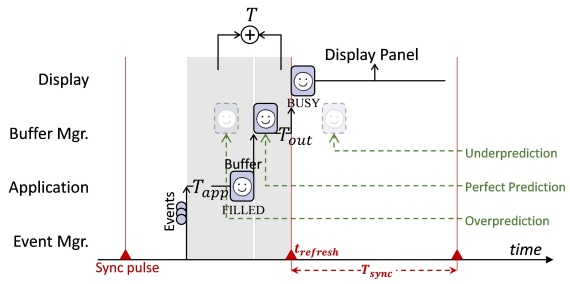


Figure 5: JITT removes synchronization in the event and buffer managers. It decides when the event manager delivers events to the application so that the buffer manager would deliver the buffer filled by the application right before the next display refresh. To do so, it must predict how long it will 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}.

to externalize, with average being $(T + 0.5 \cdot T_{sync})$. This is illustrated by the perfect prediction path 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 (§5.2.1). 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 path 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 path 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.

4.2 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 [53, 19] and a modern mobile display trend [25, 52]: an in-display memory from which the display panel reads pixels, not directly receiving from the composer. 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 [25].

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.

4.2.1 PAR: Position-Aware Rendering

To support legacy applications and displays, PAR allows an application to write in the 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} of which a prediction is available from JITT as described in §4.1. Like in JITT, underprediction is more harmful than overprediction in PAR: underprediction risks tearing effects while overprediction only decreases latency reduction.

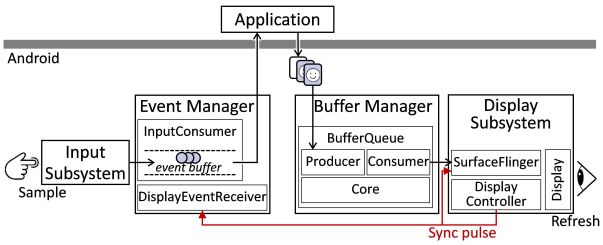


Figure 6: Android's I2D path implementation

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 §2.3. 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 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 § 5.2.2 and 6.4.4, respectively.

5 Implementation

We first describe Android's I2D path implementation, which is summarized by Figure 6, and then describe our prototype implementation of POLYPATH with two path designs, i.e., Presto and Android legacy, using Android 5 (Lollipop).

In Android 5, the event manager includes a library `libinput`, and a sync pulse receiver `DisplayEventReceiver` in Android runtime li-

brary `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: `BufferQueueCore`, `BufferQueueProducer`, and `BufferQueueConsumer`. `BufferQueue` is indirectly synchronized with the sync pulse by responding to requests from the display subsystem. `BufferQueue` is not in an application's address space; however, each application window has its dedicated `BufferQueue`. 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.

5.1 Implementing POLYPATH

Implementing POLYPATH involves three major parts: the path manager, `ApplyPath()`, and path binding. The implementation includes about 740 lines of C++ and 100 lines of Java codes.

We implement the path manager as an Android system service, which is a thread in the same address space as `SurfaceFlinger`. The event manager, however, is in the application's address space; thus, the path manager communicates with it through a Unix socket named with the application's pid.

We introduce a new class `Polypath` into Android SDK; `Polypath` exports the asynchronous path changing API `ApplyPath(app_name,path_name)`. When `ApplyPath()` is invoked, it sends `app_name` and `path_name` to the path manager through a named Unix socket. Then, the path manager communicates with the event and buffer managers to apply the path according to the design described in §3.

To implement paths and binding between path and application, we leverage that both data, i.e. events and buffer handles, and control, i.e. sync pulse, of Android's I2D path are handled by a series of function calls. For example, `InputConsumer` delivers input events using a chain of function calls, `consume()`→`consumeBatch()`→`consumeSamples()`. By changing these function calls, we can implement many path designs with various tradeoffs. For example, one can add interpolated/predicted events, drop events, or even generate virtual sync pulses to emulate 120 Hz display. We implement Presto in the same way.

To allow multiple path designs to coexist, we replace these functional calls with function pointers. By default, these pointers point to the corresponding function calls of Android's legacy implementation. When applying a new path design, we simply redirect the necessary pointers. This implementation is efficient in terms of both code size and runtime overhead. Because each application has its own dedicated event and buffer managers, one obvious alternative is to implement the event and buffer managers for each path and create an instance for each application that asks for the path. For example, we could implement a version of `libinput` for each path and an application will simply link to the corresponding version. This mechanism, however, has two disadvantages. First, it adds to software bloat because sharing common code between paths is hard, requiring major refactoring of Android's event and buffer managers. Moreover, it incurs high overhead when an application changes its path, e.g., that of unlinking and linking `libinput`.

5.2 Implementing Presto

Presto, as a path the POLYPATH supports, modifies the bindings in the event and buffer mangers for JITT and PAR. The implementation includes 991 lines of C++, 35 lines of Java, and 212 lines of Linux kernel codes.

5.2.1 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.

5.2.2 PAR: Position-Aware Buffer Manager

We implement PAR by modifying the buffer manager (`BufferQueue`) and Android's ION memory manager in the kernel. Recall that when the application requests a buffer, PAR responds with the 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 BUSY_{disp} buffer at $221\,\text{M}$ pixels per second.

If the application does not already provide information about the dirty region, e.g. via an SDK like [25], 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.

6 Evaluation

Using the prototype, we seek to answer the following questions regarding POLYPATH and Presto.

1. What is the overhead of POLYPATH?
2. How effective is Presto in reducing latency? how much does each of its two key techniques contribute?
3. Is its effectiveness orthogonal to that of other popular latency reduction technique, namely event prediction [2, 34, 36, 50, 58]?
4. What tradeoffs does Presto make, in terms of power consumption and the visual goals dear to the legacy path design?
5. How do users evaluate Presto?

6.1 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 $1.9\,\text{mm}$ [12], instead of finger, to find out the touched position with high accuracy.

Benchmarks: We evaluate Presto with both legacy applications and an in-house application. Since the effect of the latency is clearer in drawing applications, we select ten drawing applications, the top five each from the Drawing & Handwriting and Calligraphy categories of the Google Play Store on Jan 26, 2016. Some of the top applications only provide instructions for calligraphy without drawing facilities; we replaced them by the applications ranked next. 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). For these applications, we measure the latency using the indirect method presented in §6.2.1.

We also employ an in-house application because its latency can be directly measured with a more accurate method described in §6.2.2. The application uses OpenGL ES 2.0 to draw a 115×115 square and a horizontal line on a touched position. As the pen moves, it drags the square and line with it. We have implemented the application for both Android and iOS and will make both implementations open-source. The in-house application is valuable for three reasons. (*i*) First, it allows us to understand the accuracy of the indirect measurement of legacy applications. (*ii*) Second, it allows 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. and *(iii)* Because the application has bare minimum functionality for touch interaction, it allows us to better understand the power overhead of Presto.

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 $68\,\text{mm}$ per second, with a standard deviation of 12. All traces will be made available online.

6.2 Latency Measurement

Measuring the end-to-end latency of touch interaction is nontrivial because neither the starting point, the moment of a physical touch, and the end point, the moment of display externalization, can be observed by software running in the mobile device. Below we present two measurement methods used in our evaluation of Presto. 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. In our evaluation, 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.

6.2.1 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 setup with a microcontroller and two light sensors (API PDB-C142, response time: $50\,\mathrm{us}$): the microcontroller continuously polls the sensor output at $1\,\mathrm{KHz}$. We place the first light sensor besides the screen and shoot a laser beam from the other side. When the stylus pen crosses the laser beam, the light sensor detects it and changes its output; the microcontroller detects the change 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 directly 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: $28.0 \pm 1\,\mathrm{ms}$.

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

6.2.2 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.

We note that the camera also introduces errors due to its limited frame rate. We use a Nikon D5300 camera with 60 Hz frame rate and 1/500 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 for the in-house benchmark with stock Android, Presto (JITT) and Presto (JITT+PAR). The direct and indirect 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 achieved by Presto.

6.3 Overhead of POLYPATH

We measure the overhead of POLYPATH in terms of application launching delay and path switching delay on the ten drawing benchmarks applications (§6.1). When an application is launched, POLYPATH imposes the extra overhead to acquire the application's path preference from `PackageManager` and to bind the path to the application. Our measurement shows that the extra delay is between 13.1 to $34.5\,\mathrm{ms}$, which is negligible for the launch delays of many $100\,\mathrm{ms}$ for legacy applications.

The path changing API `ApplyPath()` is asynchronous in order to ensure the path integrity (§3). There is a delay between when `ApplyPath()` is invoked and when the new path is in place. Note this delay is not part of the latency of the I2D path. We estimate the delay to be between $0\,\mathrm{ms}$ and $3 \cdot T_{sync}$. The worst delay happens when the path changes from the legacy design to Presto and `ApplyPath()` is called when the event manager has only one event in its buffer. In this case, the event manager will wait almost T_{sync} to buffer more events and deliver them to the application, and the buffer manager will wait another $2 \cdot T_{sync}$ to externalize two frames, one that the application is currently drawing and the other resulting from the buffered events. We measured the time from when the API is called to when the buffer manager finishes the path change, which is the completion of the path change procedure (§3). Our measurements confirmed the above estimation with 1000 path changes each when the I2D path is active and inactive, respectively. When the I2D path is active, the average and standard deviation are $20.5\,\mathrm{ms}$ and $10.1\,\mathrm{ms}$, respectively; when inactive, they are $0.27\,\mathrm{ms}$ and $0.08\,\mathrm{ms}$, respectively.

6.4 Latency Tradeoff by Presto

We next answer the three questions about the latency reduction, its orthogonality and the tradeoffs by Presto. The measurement shows that Presto with JITT only and with JITT and 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 and will significantly improve user experience and performance according to both the literature [10] and our own experience and user study. 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

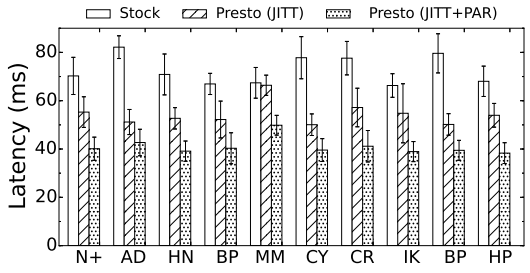


Figure 7: Presto consistently reduces the latency of the legacy benchmarks, by $32\,\mathrm{ms}$ *on average.*

touch prediction of $30\,\mathrm{ms}$, Presto is able to reduce the latency of our in-house application below $10\,\mathrm{ms}$.

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

Figure 7 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 [10] showed that even a small latency reduction, i.e. $8.3\,\mathrm{ms}$, brings a perceptible effect in touchscreen interactions. This reduction is larger than average latency caused by the coarse-grained I2D path design, i.e., $26.6\,\mathrm{ms}$ (§2.2). This is because when an application occasionally fails to 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§6.5.

Notably, different benchmarks see different amount of latency reduction 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 $17.6\,\mathrm{ms}$ only. Our analysis reveals that this is because its average T_{app} is the longest among all benchmarks. As a result, it has the least amount of latency due to synchronization and gives Presto the least opportunity.

6.4.2 Presto beats iPad Pro

Using our in-house 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: it doubles the input sampling rate for Apple Pencil [1], from 120 Hz to 240 Hz, and the iOS SDK provides predicted events for the next frame ($16\,\mathrm{ms}$), a technique called *touch prediction* [58]. Because neither technique is available on Android, we measure the in-house application on iPad Pro with four configurations as reported in the right half of Figure 8: 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 $42.9\,\mathrm{ms}$. Impressively, Presto is able to reduce Android's latency to even lower, $33.0\,\mathrm{ms}$, even without fast input sampling or touch prediction.

6.4.3 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 and the latter primarily reduce latency resulting from the input hardware. With our in-house application, we implement touch prediction that predicts into the future from 0 to $32\,\mathrm{ms}$. Presto reduces the latency by eliminating the synchronization points. Figure 9 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 $0\,\mathrm{ms}$. Clearly, for touch prediction of various time, Presto demonstrates almost the same effectiveness in latency reduction. Interestingly, Presto with touch prediction of about $30\,\mathrm{ms}$ is able to reduce the average latency below $10\,\mathrm{ms}$, a rather remarkable achievement by a software-only solution.

6.4.4 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 §6.5, 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 10, 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 §2.3, 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 §4.2.1, 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 10 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

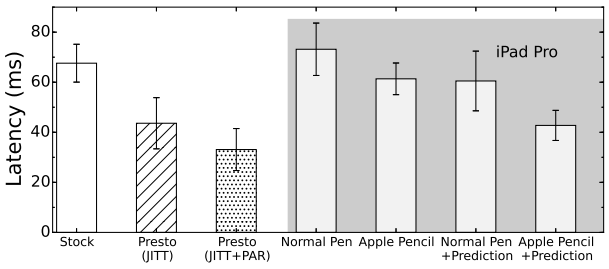


Figure 8: Presto improves the latency of our in-house application on Nexus 6 below that of iPad Pro even with Apple Pencil and touch prediction.

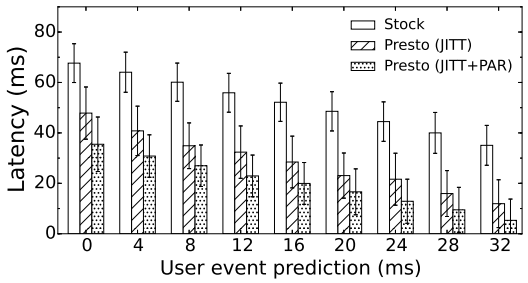


Figure 9: Latency of Presto plus touch prediction for our in-house application: the effectiveness of Presto is complementary to that of touch prediction. X axis is the time into the future predicted.

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 [41] 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.,[19], already make this information available via an API, e.g., `invalidate(Rect dirty)`. With such APIs, this overhead would be eliminated.

6.5 User Evaluation

When we try the benchmarks with Presto, it is visually obvious that Presto reduces latency significantly. None of us 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.

Participants: 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.

Procedure: 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 $240\,\mathrm{Hz}$. We plan to release the video clips in compliance with our institutional review board (IRB) approval.

Quantitative findings: Figure 11 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.[1] For Autodesk (AD), 10 out of 11 participants considers Presto is faster. This corroborates the measurement presented in Figure 7, which shows Audtodesk (AD) sees the largest latency reduction amongst the five. To our surprise and puzzle-

[1] We emphasize that the participants do not know which device has Presto. The identity is only used in our data analysis and presentation.

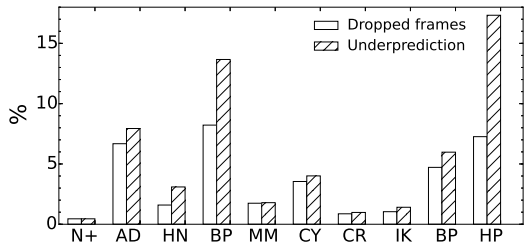


Figure 10: Presto occasionally experiences frame drops and underprediction.

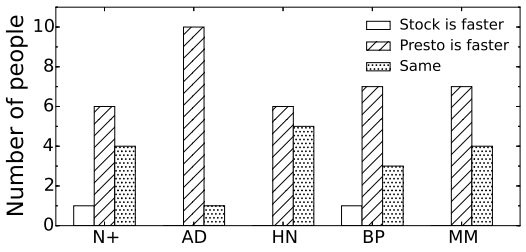


Figure 11: Number of participants answering Question (i) in each of the three ways: which device is faster: A, B or same?

ment, 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.

Qualitative findings: 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.

7 Related Work

There is a large body of literature from the systems community that reduces interaction latency by proper OS design. To the best of our knowledge, POLYPATH is the first in the public domain that serves different I2D path designs to different interactions; Presto is the first I2D path design that achieves low latency by eliminating synchronization and buffer atomicity in I2D path.

Resource Management for Low Latency: A faster computer system reduces the application execution time ($T_{app} + T_{out}$) (§2.2). The OS can also favor interactive applications in resource management to reduce their ($T_{app} + T_{out}$) [14, 31, 13, 18, 60, 62]. Many have explored the use of cloud resources to improve the interactive performance of mobile applications, e.g., [24, 23]. These solutions 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 for Low Latency: 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 the virtual reality with the head mounded display. To compensate for prediction errors, researchers have explored speculative execution [37] and post image processing [38]. All these solutions, as discussed in §6.4.3, are complementary to Presto, and can be implemented as their own paths in POLYPATH.

Specialized Hardware for Low Latency: As part of a testbed for studying touch latency, Ng et al. report an ultra-low latency touch system [44, 43] that achieves a latency as short as $1\,\mathrm{ms}$. The system employs a proprietary touch sensor with a very high sampling rate ($1\,\mathrm{KHz}$), FPGA-based low-latency ($0.1\,\mathrm{ms}$) data processing, and an ultra-high speed digital light projector ($32\,000\,\mathrm{fps}$). 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 without any delay. The game processes events without waiting for a sync pulse; when the game is in the middle of processing events, it buffers the events. Similarly, the buffer manager swaps graphics buffers 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 [51]. JITT avoids it by swapping

graphics buffers 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. In contrast, the vsync-off mode blindly ignores the sync pulses. Furthermore, simply disabling vsync on mobile devices is not feasible because display controllers on SoCs [56, 46] swap buffers only at a sync pulse regardless of when an application finishes rendering. Suppose that an application generates a frame $1\,\mathrm{ms}$ before a sync pulse and another application generates a frame $5\,\mathrm{ms}$ before a sync pulse. Regardless of when the rendering finishes, the two frames will be swapped and displayed at the next sync pulse. When a display controller swaps buffers only on a sync pulse, an application should generate multiple frames within a sync pulse period in order to reduce latency. Generating multiple frames within a sync pulse period leads to higher GPU power consumption on a mobile device [5, 45, 61] as well as more memory usage.

Nvidia's G-Sync [47] 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.

Latencies in VR Systems: State-of-the-art tethered VR systems [27, 49] have latencies between 20 and $22\,\mathrm{ms}$, much lower than those on mobile systems. However, the tethered systems are very different from mobile systems in hardware and software. They have much faster and more power-hungry hardware: inertial sensors with high sampling rates (e.g., $1\,\mathrm{KHz}$ [35]) and low latencies (e.g., $2\,\mathrm{ms}$ [48]), powerful GPUs, and higher display refresh rates (e.g., 90 or $120\,\mathrm{Hz}$). Their software takes away the composer and remove one T_{sync} period from the I2D path because VR systems has only one foreground application at a time.

Importantly, indirect input devices used in VR controllers make users less sensitive to the latency [10]. However, direct input devices such as see-through displays used in AR systems [39, 15] and touchscreens, which this paper focuses on, manifest latency as a spatial gap and require a lower latency. Presto with POLYPATH may reduce latency from AR and VR systems; however, the latency reduction will be ineffective because the average latency due to the coarse-granularities is smaller than (1) and the latency caused by the input hardware is much smaller, too.

Finally, the focus of this work, the I2D path, is reminiscent of the *path* abstraction that represents data and control flows crossing layered architectures [28, 3, 42, 17, 32, 16]. Despite the conceptual similarity, the I2D path is unique in its system context, its periodical pace and its concern with latency rather than throughout.

8 Concluding Remarks

In this work, we present POLYPATH, an operating system design that supports multiple tradeoffs for interaction latency, and Presto, an I2D path design that halves the latency by judiciously allowing frame drops. POLYPATH exports an asynchronous API that allows an unmodified legacy application to changes its path design with the guaranteed path integrity, independently from other applications.

Presto is able to reduce the average latency of the drawing benchmarks tested from about $70\,\mathrm{ms}$ to $40\,\mathrm{ms}$. *Where does the rest of latency come from?* Our investigation has pointed to the input hardware, which contributes about $30\,\mathrm{ms}$ in state-of-art 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 [33]. This latency can be reduced in two ways. One, exemplified by Apple Pencil, is to increase the input sampling rate, as shown in Figure 8. The more effective way, however, is touch prediction, as exemplified by iOS 9, as shown in Figure 8 and Figure 9.

ACKNOWLEDGEMENTS

This work was supported in part by NSF Award CNS #1422312.

References

[1] Apple. Apple pencil. http://www.apple.com/apple-pencil/, 2015.

[2] T. Asano, E. Sharlin, Y. Kitamura, K. Takashima, and F. Kishino. Predictive interaction using the delphian desktop. In *Proc. Ann. ACM Symp. User Interface Software & Technology (UIST)*, 2005.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 1995.

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

[5] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao. Rethinking energy-performance trade-off in mobile web page loading. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 14–26. ACM, 2015.

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

[7] M. Claypool and K. Claypool. Latency and player actions in online games. *Commun. ACM*, Nov. 2006.

[8] M. Claypool and K. Claypool. Latency can kill: Precision and deadline in online games. In *Proc. Ann. ACM Int. Conf. Multimedia Systems*. ACM, 2010.

[9] 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.

[10] 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 Conf. Human Factors in Computing Systems (CHI)*, 2015.

[11] 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.

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

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

[14] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer. Using latency to evaluate interactive system performance. In *Proc. USENIX Conf. Operating Systems Design & Implementation (OSDI)*, 1996.

[15] Epson. Moverio. `http://www.epson.com/Moverio`.

[16] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Proc. USENIX Annual Technical Conference (ATC)*, 2002.

[17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 1997.

[18] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on a commodity OS. In *Proc. USENIX Conf. Operating Systems Design & Implementation (OSDI)*, 2002.

[19] Google. Glsurfaceview. http://developer.android.com/reference/android/opengl/GLSurfaceView.html.

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

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

[22] Google. Parcel. `https://developer.android.com/reference/android/os/Parcel.html`.

[23] 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 Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2015.

[24] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code offload by migrating execution transparently. In *Proc. USENIX Conf. Operating Systems Design & Implementation (OSDI)*, pages 93–106, 2012.

[25] M. Ham, I. Dae, and C. Choi. LPD: Low power display mechanism for mobile and wearable devices. In *Proc. USENIX Annual Technical Conference (ATC)*, 2015.

[26] P. Henderson and J. H. Morris Jr. A lazy evaluator. In *Proc. ACM SIGACT-SIGPLAN Symp. Principles on Programming Languages (POPL)*, 1976.

[27] HTC. Htc vive. `https://www.htcvive.com`.

[28] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Software Engineering*, 1991.

[29] 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 Conf. Human Factors in Computing Systems (CHI)*, 2014.

[30] R. Jehl. Le retard tactile de l'écran de 21 smartphones et tablettes. http://www.lesnumeriques.com/telephone-portable/reactivite-tactile-ecran-21-smartphones-tablettes-n29229.html, 2013.

[31] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 1997.

[32] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 2000.

[33] S. Kolokowsky and T. Davis. *Not All Touchscreens are Created Equal - How to ensure you are developing a world class touch product*. Planet Analog, 2010.

[34] E. Lank, Y.-C. N. Cheng, and J. Ruiz. Endpoint prediction using motion kinematics. In *Proc. ACM Conf. Human Factors in Computing Systems (CHI)*, 2007.

[35] S. M. LaValle, A. Yershova, M. Katsev, and M. Antonov. Head tracking for the Oculus rift. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2014.

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

[37] 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 Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2015.

[38] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. In *Proc. Symp. Interactive 3D graphics (I3D)*. ACM, 1997.

[39] Microsoft. Hololens. `https://www.microsoft.com/microsoft-hololens`.

[40] R. B. Miller. Response time in man-computer conversational transactions. In *Proc. ACM Fall Joint Computer Conference*, 1968.

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

[42] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proc. USENIX Conf. Operating Systems Design & Implementation (OSDI)*, 1996.

[43] 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 Conf. Human Factors in Computing Systems (CHI)*, 2014.

[44] A. Ng, J. Lepinski, D. Wigdor, S. Sanders, and P. Dietz. Designing for low-latency direct-touch input. In *Proc. Ann. ACM Symp. User Interface Software & Technology (UIST)*, 2012.

[45] K. W. Nixon, X. Chen, H. Zhou, Y. Liu, and Y. Chen. Mobile gpu power consumption reduction via dynamic resolution and frame rate scaling. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*, 2014.

[46] nVidia. Tegra k1 technical reference manual.

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

[48] Oculus VR. Building a sensor for low latency vr. `https://www3.oculus.com/en-us/blog/building-a-sensor-for-low-latency-vr`.

[49] Oculus VR. Oculus rift. `https://www3.oculus.com/en-us/rift`.

[50] P. T. Pasqual and J. O. Wobbrock. Mouse pointing endpoint prediction using kinematic template matching. In *Proc. ACM Conf. Human Factors in Computing Systems (CHI)*, 2014.

[51] T. Petersen. Gpu boost 3 and sli, May 7, 2016.

[52] E. Petillon. Demystify DSI I/F. Texas Instruments, 2012.

[53] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 1.0), 1992.

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

[55] D. Stoza. libui/libgui: Fix errors in parceling. Commit ID: eea6d682b8b0f7081f0fe8fab8feadb16e22b30b.

[56] Texas Instruments. Omap4460 multimedia device silicon: Trm.

[57] R. Thuret. Samsung galaxy note 3 : que valent son cran amoled et son appareil photo. http://www.lesnumeriques.com/telephone-portable/samsung-galaxy-note-3-que-valent-son-ecran-amoled-son-appareil-photo-a1740.html, 2013.

[58] P. Tsoi and J. Xiao. Advanced touch input on iOS: Increasing responsiveness by reducing latency. The Apple Worldwide Developers Conference (WWDC), 2015.

[59] 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.

[60] L. Yan, L. Zhong, and N. Jha. Towards a responsive, yet power-efficient, operating system: A holistic approach. In *Proc. IEEE Int. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunications Systems(MASCOTS)*, 2005.

[61] Y. Yan, S. He, Y. Liu, and L. Huang. Optimizing power consumption of mobile games. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, pages 21–25. ACM, 2015.

[62] 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 Conf. Operating Systems Design & Implementation (OSDI)*, 2008.