Replaced by http://github.com/wicg/event-timing

Event Timing Web Perf API

STATUS: Seeking feedback.

tdresser@
Last updated: July 24, 2017

Background

Web developers currently have little insight into what causes latency when handling events.

This document provides a "Minimal Proposal" which provides developers insight into the cases which are most often slow. This proposal does not address latency of input which isn't blocked on the browser's main thread. Looking forward, more and more input latency will be due to threads other than the main thread, and we'll eventually need to extend this API to cover these cases, such as animation worklet and offscreen canvas.

Minimal Proposal

A previous proposal grew too broad in scope. This proposal explains the minimal API required to solve the following key use cases:

1. Measure event handler / default action duration.
2. Correlate input with slow frames.
    - Including input without handlers, such as input triggering hover.
3. Measure impact of event handlers on scroll performance
    - For scrolls blocked by main thread work.

A polyfill roughly implementing part of this API can be found here.

In order to accomplish these goals, we introduce:

```
interface PerformanceEventTiming : PerformanceEntry {
    // The type of event dispatched. E.g. "touchmove".
    // Doesn't require an event listener of this type to be registered.
    readonly attribute DOMString name;
    // "event".
    readonly attribute DOMString entryType;
    // The event timestamp.
    readonly attribute DOMHighResTimeStamp startTime;
    // The time the first event handler or default action started to execute.
    // startTime if no event handlers or default action executed.
    readonly attribute DOMHighResTimeStamp processingStart;
    // The time the last event handler or default action finished executing.
    // startTime if no event handlers or default action executed.
    readonly attribute DOMHighResTimeStamp processingEnd;
    // Zero.
    readonly attribute DOMHighResTimeStamp duration;
    // Whether or not the event was cancelable.
    readonly attribute boolean cancelable;
    // Whether or not the event contributed to a user scroll.
    readonly attribute boolean eventCausedScroll;
    // Whether or not a commit was pending at processingEnd.
    readonly attribute boolean eventHasCommit;
    // If a commit was pending at processingEnd, the time that commit occurred.
    // Otherwise, 0.
    readonly attribute DOMHighResTimeStamp commitTime;
};
```

Correlating with Frame Timing

The startTimeUntilFrameDuration field provides information about when an associated commit occurred, if one did. However, in the future, we'd like to be able to associate this with when pixels actually hit the screen. The Frame Timing API will eventually surface more accurate display timestamps, and will also include the frame commit time. startTimeUntilFrameDuration will enable us to correlate event processing entries with frame timing entries, letting us measure from input time until pixels hit the screen.

When should we dispatch PerformanceEventTiming entries?

Proposal:

Let's dispatch `PerformanceEventTiming` for all events for which $max(processingEnd,\ startTime\ +\ startTimeUntilFrameDuration)\ -\ eventQueued\ >\ 50ms$

This considers an event to start the moment it's blocked on the main thread, and an event to end when it's associated frame is committed, if one exists, and when the event handlers and default action are complete if no associated frame exists.

Alternatives:

- We could batch these entries, and dispatch one per event, but this is likely to incur too high a performance overhead, and may constrain what information we're allowed to include in the Entry due to privacy concerns.
- We could dispatch these entries if the event handlers or the default action ran longer than some threshold. This would miss a variety of cases however, such as when event handlers dirty style or layout, and the style or layout recalculation is expensive.

What about input triggering multiple DOM event types?

Proposal:

- Report one entry per DOM event.
    - For events which have no listeners, we report one entry per DOM event which would have been dispatched had there been listeners.
    - For nested elements which all have, for example, `touchmove` event handlers, only one entry is reported, despite there being many listeners.
    - For input which triggers multiple DOM events, such as a touch pointer release triggering `touchend`, `pointerend` and `click`, many entries may be dispatched for a single user input.

Alternatives:

- Report one entry per logical user input.
    - We could dispatch less redundant information if we only reported one entry per logical user input. For example, a touch pointer release would report a single

entry, instead of reporting `touchend`, `pointerend` and `click`. The primary advantage of using the existing DOM event types is simplicity - developers already understand DOM event types, specs already include the notion of DOM event types, and this allows us to assume that all event listener invocations occur in contiguous blocks.

- Report one entry per event listener invocation.
    - Developers will want to monitor the total amount of work done per event, so we should perform this aggregation step for them.

Is a Polyfill Good Enough?

Polyfilling this runs into a few problems.

1. Measure event handler duration.
    - It's possible to polyfill this, but it's tricky to all combine timing data associated with a single DOM event. See a possible solution here. This solution doesn't have adequate performance, as it requires replacing addEventListener, and measuring the time at the end of every event listener invocation.
2. Correlate input with slow frames.
    - This is possible to polyfill for input with event handlers, but impossible for input without event handlers. Adding event handlers for all input types has unacceptable performance overhead.
3. Measure impact of event handlers on scroll performance
    - We can get close to polyfilling this (with the above caveats), by identifying cancellable events of types which could potentially trigger scroll, which executed during a frame in which a scroll occurred. We could remove `causedScroll` from `PerformanceEventTiming` and require users to correlate this with scroll events, but including it makes this a fair bit easier for consumers of this API, and will result in slightly higher quality results.

But what about X?

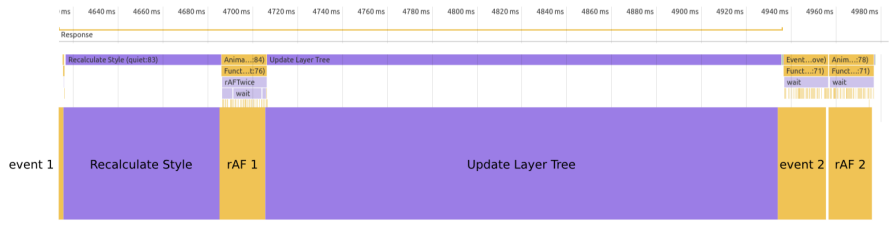
The Long Tasks API

The long tasks API provides some of the value of this API, but isn't adequate. The work done in response to an event is often split between multiple tasks. First, the event handlers are executed, which often dirties layout and style. Then when we go to produce a frame, rAF is executed, potentially based on input handled previously, and we perform style and layout. That style and layout needs to be included as part of the cost of event handling, and this isn't doable using the long tasks API. In order to approximate the user perceived latency of an event, we need to include the time taken to draw the frame, and the long tasks API doesn't let us do that.

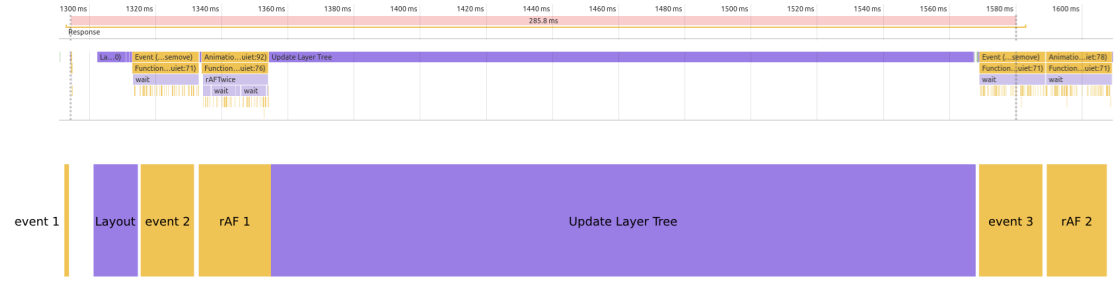
Here are two examples of events being processed where the handling time is very short, but the total time taken to display the result of the input is very long. The long tasks API would notify about the long style recalc and layer tree update, but wouldn't correlate with the event. In order to understand the total work done in processing the event, we need to know the time taken until we've produced a frame.

output.jsbin.com/xoxivu/quiet

Style only



Layout Only



Even if the long tasks API did give us information about any events which triggered this long task, it still wouldn't solve the problem of correlating all work associated with an event, and wouldn't threshold on the actual event latency. The event timing API needs to dispatch a performance entry if the total time spent processing the event on the main thread exceeds some threshold.

In addition, this API provides additional context on the event in question.

Composited Scrolling

Composited Scrolling which doesn't block on the main thread should be essentially equivalent to other composited animations, and should eventually be addressed by extensions to the Frame Timing API. The changes a developer can make to fix high compositing overhead are equivalent

for scrolling and other composited animations. Insight into the events which caused the scrolling isn't very valuable in this context.

Cases where we only want to monitor one event type

We may want to support some way of filtering which DOM event types are monitoring at Performance Observer registration time. This could be achieved by extending PerformanceObserverInit.

Cases where $THRESHOLD ms is too much Latency to ignore

The Frame Timing current proposal states "the user agent is allowed to set and exercise own thresholds for delivery of slow frame events." We could eventually enable configuring the long frame threshold, or we could be smart about picking when to report long frames, based on the context. We can use the same approach for the Event Timing API.

Cases where we want more or different timestamps

There are lots more useful timestamps, which should be added to these performance entries in the future. Most of this proposal can be incrementally extended by adding additional timestamps to the frame or event performance entries. The hardest thing to iterate on is the duration that we threshold on for deciding whether or not to fire a long frame entry.

The current proposal suggests using main thread frame time (or event handling time, if no frame was produced). There are reasonable arguments to be made that we should instead threshold on our best guess of how long the frame took to produce, all the way up until the frame hits the glass. In order to minimize spec churn, I think it makes sense to go with the main thread time for now, and we can consider ways of dealing with cases which fail because we're using the main thread frame time in the future. If we tune the threshold correctly, as long as we add the estimated glass time to the Frame Timing performance entry, thresholding on the main thread time will be fine.

A few additional pieces of information we should add in the future:

- Frame Timing
    - Estimated Glass Time
    - rAF handler start/end.
    - Time spent compositing
- Event Timing
    - Long tasks V2 style attribution for event handlers.

:visited attacks

This proposal opens up a new approach for sniffing a user's history via :visited. To accomplish this, an attacker would:

- Add a link to $SITE
- Give the link a :visited:hover style which paints
- Avoid giving the link a :hover style which paints
- Give the link element a long (>50ms) mousemove event handler.
- Add an event timing performance observer.

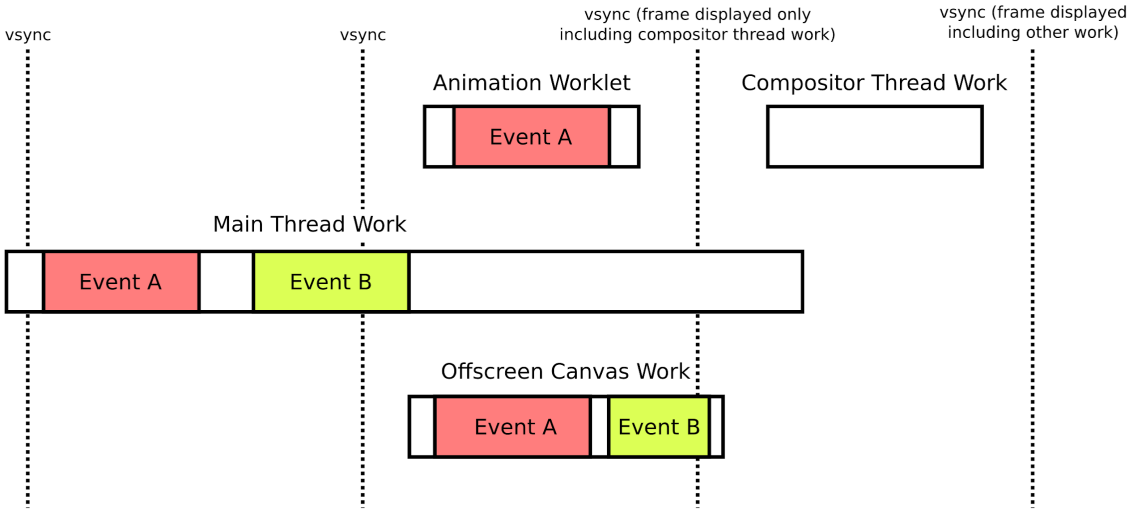
When a performance entry for this mouse move event comes in, then if there's an associated commit, then the user has previously visited $SITE.

One way to avoid this issue would be to always commit when hovering an object which has a :hover:visited style.

Planning for the Future

The current Frame Timing proposal, and the minimal proposal above both only pay attention to the main thread. This is fine for V1, but we want to ensure these proposals have clear paths to extending to include work done on other threads.

In general, each thread which does work related to frame production has a separate contiguous block of time dedicated to processing each frame. During each frame, some number of events may be processed on each thread, during that thread's block of time allocated to processing that frame. Each frame is displayed at a single time, providing a clear way of associating the work done to produce a frame across all threads.



When we get to the point where we allow input handling on other threads, we'll need to extend the minimal proposal to support monitoring the performance of this input. What this looks like will depend on the APIs we use for exposing input, but here's a plausible approach, demonstrating that the minimal approach proposed above can be extended to support these use cases.

If we modify PerformanceObserverInit to allow passing in a worker or worklet which is receiving input or producing frames, then we could report event and frame timing data in that context.

```
dictionary PerformanceObserverInit {
    ...
    optional (Worker or Worklet)? context;
};


interface PerformanceEventTiming : PerformanceEntry {
    ...
    readonly attribute (Worker or Worklet)? context;
}

interface PerformanceFrameTiming : PerformanceEntry {
    ...
    readonly attribute (Worker or Worklet)? context;
}

const performanceObserver = new PerformanceObserver(...);
performanceObserver.observe({entryTypes: ["frame", "event"], context: worker};
```

The frame and event entries dispatched to this observer will include a `context` attribute, indicating what context the entry timestamps are relative to. Frame timestamps and event timestamps would be relative to the context in question.