Debug performance in the field

- APIs for attribution and debugging
  - CLS
    - Get layout shift attribution
  - LCP
    - Identify the LCP candidate element
  - FID
    - Identify the FID target element and event type
  - INP
- Usage with the web-vitals JavaScript library
- Report and visualize the data
  - The Web Vitals Report tool
- Summary

Google currently provides two categories of tools to measure and debug performance:

- **Lab tools:** Tools such as Lighthouse, where your page is loaded in a simulated environment that can mimic various conditions (for example, a slow network and a low-end mobile device).
- **Field tools:** Tools such as Chrome User Experience Report (CrUX), which is based on aggregate, real-user data from Chrome. (Note that the field data reported by tools such as PageSpeed Insights and Search Console is sourced from CrUX data.)

While field tools offer more accurate data—data which actually represents what real users experience—lab tools are often better at helping you identify and fix issues.

CrUX data is more representative of your page's real performance, but knowing your CrUX scores is unlikely to help you figure out *how* to improve your performance.

Lighthouse, on the other hand, will identify issues and make specific suggestions for how to improve. However, Lighthouse will only make suggestions for performance issues it discovers at

page load time. It does not detect issues that only manifest as a result of user interaction such as scrolling or clicking buttons on the page.

This raises an important question: **how can you capture debug information for Core Web Vitals or other performance metrics from real users in the field?**

This post will explain in detail what APIs you can use to collect additional debugging information for each of the current Core Web Vitals metrics and give you ideas for how to capture this data in your existing analytics tool.

APIs for attribution and debugging

CLS

Of all the Core Web Vitals metrics, CLS is perhaps the one for which collecting debug information in the field is the most important. CLS is measured throughout the entire lifespan of the page, so the way a user interacts with the page—how far they scroll, what they click on, and so on—can have a significant impact on whether there are layout shifts and which elements are shifting.

Consider the following report from PageSpeed Insights for the URL: web.dev/measure

{% Img src="image/eqprBhZUGfb8WYnumQ9ljAxRrA72/nZjd6rXrOgW5VUsm5fyx.png", alt="A PageSpeed Insights Report with different CLS values", width="800", height="587" %}

The value reported for CLS from the lab (Lighthouse) compared to the CLS from the field (CrUX data) are quite different, and this makes sense if you consider that the web.dev/measure page has a lot of interactive content that is not being used when tested in Lighthouse.

But even if you understand that user interaction affects field data, you still need to know what elements on the page are shifting to result in a score of 0.45 at the 75th percentile.

The LayoutShiftAttribution interface makes that possible.

Get layout shift attribution

The LayoutShiftAttribution interface is exposed on each `layout-shift` entry that Layout Instability API emits.

For a detailed explanation of both of these interfaces, see Debug layout shifts. For the purposes of this post, the main thing you need to know is that, as a developer, you are able to observe every layout shift that happens on the page as well as what elements are shifting.

Here's some example code that logs each layout shift as well as the elements that shifted:

```
new PerformanceObserver((list) => {
  for (const {value, startTime, sources} of list.getEntries()) {
    // Log the shift amount and other entry info.
    console.log('Layout shift:', {value, startTime});
    if (sources) {
      for (const {node, curRect, prevRect} of sources) {
        // Log the elements that shifted.
        console.log('  Shift source:', node, {curRect, prevRect});
      }
    }
  }
}).observe({type: 'layout-shift', buffered: true});
```

It's probably not practical to measure and send data to your analytics tool for every single layout shift that occurs; however, by monitoring all shifts, you can keep track of the worst shifts and just report information about those.

The goal isn't to identify and fix every single layout shift that occurs for every user; the goal is to identify the shifts that affect the largest number of users and thus contribute the most to your page's CLS at the 75th percentile.

Also, you don't need to compute the largest source element every time there's a shift, you only need to do so when you're ready to send the CLS value to your analytics tool.

The following code takes a list of `layout-shift` entries that have contributed to CLS and returns the largest source element from the largest shift:

```
function getCLSDebugTarget(entries) {
  const largestEntry = entries.reduce((a, b) => {
    return a && a.value > b.value ? a : b;
  });
  if (largestEntry && largestEntry.sources && largestEntry.sources.length) {
    const largestSource = largestEntry.sources.reduce((a, b) => {
      return a.node && a.previousRect.width * a.previousRect.height >
          b.previousRect.width * b.previousRect.height ? a : b;
    });
    if (largestSource) {
      return largestSource.node;
    }
  }
```

```
}
```

Once you've identified the largest element contributing to the largest shift, you can report that to your analytics tool.

The element contributing the most to CLS for a given page will likely vary from user to user, but if you aggregate those elements across all users, you'll be able to generate a list of shifting elements affecting the most number of users.

After you've identified and fixed the root cause of the shifts for those elements, your analytics code will start reporting smaller shifts as the "worst" shifts for your pages. Eventually, all reported shifts will be small enough that your pages are well within the "good" threshold of 0.1!

Some other metadata that may be useful to capture along with the largest shift source element is:

- The time of the largest shift
- The URL path at the time of the largest shift (for sites that dynamically update the URL, such as Single Page Applications).

LCP

To debug LCP in the field, the primary information you need is which particular element was the largest element (the LCP candidate element) for that particular page load.

Note that it's entirely possible—in fact, it's quite common—that the LCP candidate element will be different from user to user, even for the exact same page.

This can happen for several reasons:

- User devices have different screen resolutions, which results in different page layouts and thus different elements being visible within the viewport.
- Users don't always load pages scrolled to the very top. Oftentimes links will contain fragment identifiers or even text fragments, which means it's possible for your pages to be loaded and displayed at any scroll position on the page.
- Content may be personalized for the current user, so the LCP candidate element could vary wildly from user to user.

This means you cannot make assumptions about which element or set of elements will be the most common LCP candidate element for a particular page. You have to measure it based on real-user behavior.

{% Aside %} For more details on these differences, see: Why lab and field data can be different (and what to do about it) {% endAside %}

Identify the LCP candidate element

To determine the LCP candidate element in JavaScript you can use the Largest Contentful Paint API, the same API you use to determine the LCP time value.

When observing `largest-contentful-paint` entries, you can determine the current LCP candidate element by looking at the `element` property of the last entry:

```
new PerformanceObserver((list) => {
  const entries = list.getEntries();
  const lastEntry = entries[entries.length - 1];

  console.log('LCP element:', lastEntry.element);
}).observe({type: 'largest-contentful-paint', buffered: true});
```

{% Aside 'caution' %} As explained in the LCP metric documentation, the LCP candidate element can change through the page load, so more work is required to identify the "final" LCP candidate element. The easiest way to identify and measure the "final" LCP candidate element is to use the web-vitals JavaScript library, as shown in the example below. {% endAside %}

Once you know the LCP candidate element, you can send it to your analytics tool along with the metric value. As with CLS, this will help you identify which elements are most important to optimize first.

In addition to the LCP candidate element, it may also be useful to measure the LCP sub-part times, which can be useful in determining what specific optimization steps are relevant for your site.

FID

To debug FID in the field, it's important to remember that FID measures only the delay portion of the overall first input event latency. That means that what the user interacted with is not really as important as what else was happening on the main thread at the time they interacted.

For example, many JavaScript applications that support server-side rendering (SSR) will deliver static HTML that can be rendered to the screen before it's interactive to user input—that is, before the JavaScript required to make the content interactive has finished loading.

For these types of applications, it can be very important to know whether the first input occurred before or after hydration. If it turns out that many people are attempting to interact with the page before hydration completes, consider rendering your pages in a disabled or loading state rather than in a state that looks interactive.

If your application framework exposes the hydration timestamp, you can easily compare that with the timestamp of the `first-input` entry to determine whether the first input happened before or after hydration. If your framework doesn't expose that timestamp, or doesn't use hydration at all, another useful signal may be whether input occurred before or after JavaScript finished loading.

The `DOMContentLoaded` event fires after the page's HTML has completely loaded and parsed, which includes waiting for any synchronous, deferred, or module scripts (including all statically imported modules) to load. So you can use the timing of that event and compare it to when FID occurred.

The following code observes `first-input` entries and logs whether or not the first input occurred prior to the end of the `DOMContentLoaded` event:

```
new PerformanceObserver((list) => {
  const fidEntry = list.getEntries()[0];
  const navEntry = performance.getEntriesByType('navigation')[0];
  const wasFIDBeforeDCL =
    fidEntry.startTime < navEntry.domContentLoadedEventStart;

  console.log('FID occurred before DOMContentLoaded:', wasFIDBeforeDCL);
}).observe({type: 'first-input', buffered: true});
```

{% Aside %} If your page uses `async` scripts or dynamic `import()` to load JavaScript, the `DOMContentLoaded` event may not be a useful signal. Instead, you can consider using the `load` event or—if there's a particular script you know takes a while to execute—you can use the Resource Timing entry for that script. {% endAside %}

Identify the FID target element and event type

Additional potentially-useful debug signals are the element that was interacted with as well as the type of interaction it was (such as `mousedown`, `keydown`, `pointerdown`). While the interaction with the element itself does not contribute to FID (remember FID is just the delay portion of the total event latency), knowing which elements your users are interacting with may be useful in determining how best to improve FID.

For example, if the vast majority of your user's first interactions are with a particular element, consider inlining the JavaScript code needed for that element in the HTML, and lazy loading the rest.

To get the interaction type and element associated with the first input event, you can reference the `first-input` entry's `target` and `name` properties:

```
new PerformanceObserver((list) => {
  const fidEntry = list.getEntries()[0];
```

```
  console.log('FID target element:', fidEntry.target);
  console.log('FID interaction type:', fidEntry.name);
}).observe({type: 'first-input', buffered: true});
```

INP

INP is very similar to FID in that the most useful bits of information to capture in the field are:

1. What element was interacted with
2. Why type of interaction it was
3. When that interaction took place

Like FID, a major cause of slow interactions is a blocked main thread, which can be common while JavaScript is loading. Knowing whether most slow interactions occur during page load is helpful in determining what needs to be done to fix the problem.

Unlike FID, the INP metric considers the full latency of an interaction—including the time it takes to run any registered event listeners as well as the time it takes to paint the next frame after all events listeners have run. This means that for INP it's even more useful to know which target elements tend to result in slow interactions, and what types of interactions those are.

Since INP and FID are both based on the Event Timing API, the way you determine this information in JavaScript is very similar to the previous example. The following code logs the target element and time (relative to `DOMContentLoaded`) of the INP entry.

```
function logINPDebugInfo(inpEntry) {
  console.log('INP target element:', inpEntry.target);
  console.log('INP interaction type:', inpEntry.name);

  const navEntry = performance.getEntriesByType('navigation')[0];
  const wasINPBeforeDCL =
    inpEntry.startTime < navEntry.domContentLoadedEventStart;

  console.log('INP occurred before DCL:', wasINPBeforeDCL);
}
```

Note that this code doesn't show how to determine which `event` entry is the INP entry, as that logic is more involved. However, the following section explains how to get this information using the web-vitals JavaScript library.

Usage with the web-vitals JavaScript library

The sections above offer some general suggestions and code examples to capture debug info to include in the data you send to your analytics tool.

Since version 3, the web-vitals JavaScript library includes an attribution build that surfaces all of this information, and a few additional signals as well.

The following code example shows how you could set an additional event parameter (or custom dimension) containing a debug string useful for helping to identify the root cause of performance issues.

```
import {onCLS, onFID, onINP, onLCP} from 'web-vitals/attribution';

function sendToGoogleAnalytics({name, value, id, attribution}) {
  const eventParams = {
    metric_value: value,
    metric_id: id,
  }

  switch (name) {
    case 'CLS':
      eventParams.debug_string = attribution.largestShiftTarget;
      break;
    case 'LCP':
      eventParams.debug_string = attribution.element;
      break;
    case 'FID':
    case 'INP':
      eventParams.debug_string = attribution.eventTarget;
      break;
  }

  // Assumes the global `gtag()` function exists, see:
  // https://developers.google.com/analytics/devguides/collection/ga4
  gtag('event', name, eventParams);
```

```
}

onCLS(sendToGoogleAnalytics);
onLCP(sendToGoogleAnalytics);
onFID(sendToGoogleAnalytics);
onINP(sendToGoogleAnalytics);
```

This code is specific to Google Analytics, but the general idea should easily translate to other analytics tools as well.

This code also just shows how to report on a single debug signal, but it may be useful to be able to collect and report on multiple different signals per metric. For example, to debug INP you might want to collect the interaction type, time, and also the element being interacted with. The `web-vitals` attribution build exposes all of this information, as show in the following example:

```
import {onCLS, onFID, onINP, onLCP} from 'web-vitals/attribution';

function sendToGoogleAnalytics({name, value, id, attribution}) {
  const eventParams = {
    metric_value: value,
    metric_id: id,
  }

  switch (name) {
    case 'INP':
      eventParams.inp_target = attribution.eventTarget;
      eventParams.inp_type = attribution.eventType;
      eventParams.inp_time = attribution.eventTime;
      eventParams.inp_load_state = attribution.loadState;
      break;

    // Additional metric logic...
  }

  // Assumes the global `gtag()` function exists, see:
  // https://developers.google.com/analytics/devguides/collection/ga4
  gtag('event', name, eventParams);
}

onCLS(sendToGoogleAnalytics);
onLCP(sendToGoogleAnalytics);
```

```
onFID(sendToGoogleAnalytics);
onINP(sendToGoogleAnalytics);
```

Refer to the web-vitals attribution documentation for the complete list of debug signals exposed.

Report and visualize the data

Once you've started collecting debug information along with the metric values, the next step is aggregating the data across all your users to start looking for patterns and trends.

As mentioned above, you don't necessarily need to address every single issue your users are encountering, you want to address—especially at first—the issues that are affecting the largest number of users, which should also be the issues that have the largest negative impact on your Core Web Vitals scores.

The Web Vitals Report tool

If you're using the Web Vitals Report tool, it allows you to additionally report on a single debug dimension for each of the Core Web Vitals metrics.

Here's a screenshot from the Web Vitals Report debug info section, showing data for the Web Vitals Report tool website itself:

{% Img src="image/eqprBhZUGfb8WYnumQ9ljAxRrA72/Y49u3cmRD6RfAaZGCSmx.png", alt="Web Vitals Report showing debug information", width="800", height="535" %}

Using the data above, you can see that whatever is causing the `section.Intro` element to shift is contributing the most to CLS on this page, so identifying and fixing the cause of that shift will yield the greatest improvement to the score.

Summary

Hopefully this post has helped outline the specific ways you can use the existing performance APIs and the `web-vitals` library to get debug information to help diagnose performance based on real users visits in the field. While this guide is focused on the Core Web Vitals, the concepts also apply to debugging any performance metric that's measurable in JavaScript.

If you're just getting started measuring performance, and you're already a Google Analytics user, the Web Vitals Report tool may be a good place to start because it already supports reporting

debug information for the Core Web Vitals metrics.

If you're an analytics vendor and you're looking to improve your products and provide more debugging information to your users, consider some of the techniques described here but don't limit yourself to *just* the ideas presented here. This post is intended to be generally applicable to all analytics tools; however, individual analytics tools likely can (and should) capture and report even more debug information.

Lastly, if you feel there are gaps in your ability to debug these metrics due to missing features or information in the APIs themselves send your feedback to web-vitals-feedback@googlegroups.com.

□□□ #2
□ □□□ □□□ 2026-10-31 21:49:02 UTC
□ □□□ □□□ 2026-10-31 21:49:04 UTC