Measuring Chrome Responsiveness

Status: Reviewed
Author: erikchen
Reviewers: tdresser, asvitkine, wittman, pkasting, gab, robliao, skyostil
Last Updated: 2018-07-20

Note: This document is a copy. The original was reviewed here.

Objective

Create a widely applicable performance metric that reflects the quality of user experience with UI interactions.

Goals:

- Create a metric that reflects the user experience with UI interactions.
- Create a target for improvement for teams working on browser process performance.
- Create a minimum performance bar that is evaluated for all experiments [heartbeat].
- Low performance overhead.

Non-Goals:

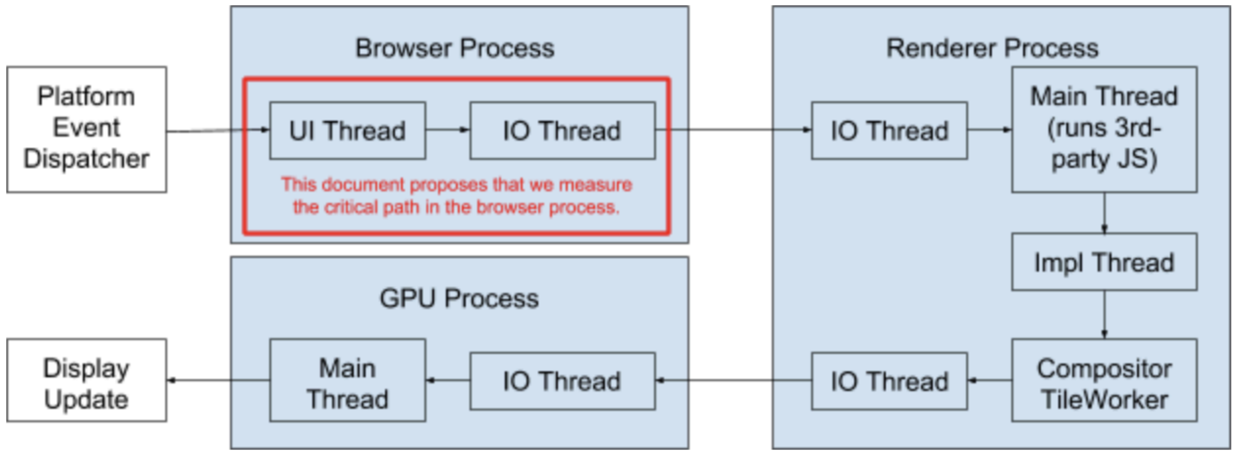
- Create a metric that reflects all responsiveness regressions.
- Create a mechanism to pinpoint regressions.
    - Like other heartbeat metrics, this metric will be used to detect regression in Finch experiments and telemetry tests. However, an isolated regression in UMA will not be actionable.
    - Both the CPU sampling profiler [wittman@], and the Chrome-Catan team are working on solutions for getting actionable information from the field. Perfetto is a distant future solution.

Necessary but not sufficient:

- This metric reflects the best possible responsiveness of user interaction. The actual latency is higher.
- A regression in this metric implies that user experience with UI interactions has regressed.
- It is possible for user experience with UI interactions to regress without regressing this metric.

Background

Very rough depiction of critical path from user input to display change, with Viz, assuming that the user clicks a button on a web page.



Overview

Chrome has many specialized performance metrics, but none of them reflect the overall performance of Chrome. As a first order approximation, we measure the execution latency of posted tasks and native events on the UI and IO threads of the browser process. This establishes a minimum bound for responsiveness. A higher execution latency for either thread, implies with high probability that Chrome will not be very responsive.

We transform this into a time-normalized measure of responsiveness. We discretize time into short intervals, and mark each interval as janky if there was a posted task or queued event whose execution latency was user-noticeable. We use this to create the metric JankyIntervalsPerThirtySeconds.

- This metric can be analyzed in aggregate to understand Chrome's responsiveness, weighted by time spent using Chrome
  - A client that spends more time using Chrome will be weighted more heavily.
- This metric can be averaged by client, and then analyzed in aggregate to understand Chrome's responsiveness, with all clients counted equally.
  - This avoids the bias of discounting clients that have a bad experience [and thus use Chrome less]

Caveats

An accurate measurement for responsiveness would need to trace the critical path of events from user input to display change. This depends on at least 4 different schedulers across 3 processes. It depends on the operating system, the graphics drivers, and the content being interacted with. While it would be possible to measure responsiveness in any given scenario, I don't believe it's possible to come up with a generic definition that works for all scenarios.

For example, LatencyInfo attempts to measure responsiveness for very specific scenarios [e.g. scroll events, mouse events]. To do so, it uses a 21 case enum [see LatencyComponentType] to record data, and emits 15 groups of histograms, several of which emit 16 histograms each [see LatencyTracker]. It doesn't cover all types of input events [e.g. macOS hotkeys that are handled by AppKit]. And there's no way to aggregate the metric to create a meaningful top-level metric.

We choose not to measure the responsiveness of the renderer or GPU processes, as the measurement is dominated by the behavior of 3rd party JS, and we want the metric to reflect responsiveness changes caused by Chrome.

On all platforms, the MessagePump of the UI thread of the browser process pulls events from a separate queue than Chrome tasks, and the event queue is given higher priority. At face value, this means that task queueing time on the UI thread does not impact the critical path for event handling. However, tasks posted on the UI thread are intended to affect the UI [e.g. updating the tab strip], so execution latency is highly likely relevant to the critical path.

Similarly, the message pump of the IO thread will process incoming IPC messages separately from the task queue. This means that the task queueing time of the IO thread does not directly affect the execution latency of incoming IPC messages. However, tasks posted to the IO thread are used for outgoing IPC, which are usually a part of the critical path.

Unlike native events on the UI thread, there is no way to measure the queueing time of incoming IPC messages on the IO thread with the current wire format. Furthermore, the execution time for handling of incoming IPC messages is almost always small, as significant work is posted to other threads. Therefore, we chose not to measure execution time of incoming IPC messages, as it would add complexity and reduce performance for little expected benefit.

For alternative methods of slicing the same data, see Appendix B.

The combination of accounting for queueing latency and execution time should account for most sources of an unresponsive thread.

Detailed Design

Definitions

Given a posted task T or native event E, define

- ExecutionLatency(T) = time_that_execution_finished - time_that_task_was_posted.
- ExecutionLatency(E) = time_that_execution_finished - time_that_event_was_created

Given a jank threshold [X = 100ms],

- A task or event is janky if ExecutionLatency(T) > X.
- 100ms is chosen to be large enough to be obviously noticeable by most users, and small enough that I expect meaningful room for improvement for Chrome.
    - This is also the threshold outlined in the rail guidelines.
    - And is independently verified to be a relevant threshold for response time.
- Note: This threshold will be configurable. We may want a lower threshold [e.g. 50ms] in the future.

We discretize time into 100ms time slices [using the same threshold as above].

- A thread is janky in a given time slice if there exists at least one janky task or event that ran, but did not start, in that time slice.
    - A 100ms task that spans two time slices will cause the second time slice to be considered janky, but not the first.
    - A 200ms task that spans three time slices will cause the second and third time slices to be considered janky, but not the first.
- Chrome is janky in a given time slice if either the UI or IO threads of the browser process were janky.

We pick a large time interval [30 seconds] and define

- JankyIntervalsPerThirtySeconds = (# of janky intervals).
- 30 seconds is large enough to avoid most effects of boundary conditions on the calculation, but small enough to meaningful for short browsing sessions, even on mobile.
    - Large session times [e.g. 1 hour] are not meaningful on mobile. See discussion.
- We will record this metric every 30 seconds.
    - Each time a task or event finishes executing on the UI, we will check to see if more than 30s has passed since the beginning of the current interval. If so, janks will be calculated and recorded for the previous interval.
    - This avoids using a 30s timer, as that would cause tasks to be posted [waking up Chrome], even if nothing else is happening.
- Partial sessions due to early session termination [mostly effects mobile], will be discarded. This loses some data, but avoids noise from up-scaling shorter intervals.

    - We will also record a second UMA count for the # of partial sessions, whose absolute counts will be compared to the counts from the primary UMA to make sure we're not losing too much data

Note: All computations [except for the final aggregation by client ID] occur on the client and metrics are sent as normal UMA histograms.

Examples

These examples count the # of janky intervals in a variety of scenarios:

1. In a theoretical no-op application, where the UI and IO thread never processes any tasks, there are no janky intervals.
2. If Chrome is behaving properly, and the UI and IO threads are never backed up, then there are no janky intervals.
3. There is a single task on the UI thread that takes 3000ms. This creates 30 janky intervals.
4. There are 5 tasks on the UI thread that each take 600ms. This creates 30 janky intervals.
5. There is a task that reposts itself on completion, up to 10 times. It takes 50ms to run. This creates 0 janky intervals.
6. There are 10 tasks posted back-to-back, each takes 50ms to run. This creates 5 janky intervals.
7. Chrome is constantly posting tasks, creating a backlog, such that all events take over a second to dequeue. This creates 3000 janky intervals.
8. Chrome spends 30 minutes behaving properly like (2), and 30 minutes behaving poorly like (7). This creates 1500 janky intervals.
9. Every 10 seconds, there are 1000 tiny tasks [each taking 1ms] that are posted back-to-back. This creates 9 janky intervals every 10 seconds, for a total of 27 janky intervals in 30 seconds.

Performance

The main performance overhead will come from:

- Calling TimeTicks::Now() at queueing time and finished-execution time for each task.
    - This is used to compute TaskScheduler.TaskLatencyMicroseconds.*, but that isn't used for MessageLoopTaskRunner.
    - Overhead is measured here.
- Calling Time::Now() at the end of native event execution, to compare against the creation timestamp of native events.

Future Updates

Task Prioritization

There is a plan to add task prioritization to the UI and IO threads. Once this is implemented, we may wish to consider adding a new metric which only measures execution latency of high-priority tasks.

Measuring Jank in Other Processes

- Processing incoming IPCs on the IO thread of the GPU process is always a part of the critical path for frame synchronization, and the effects are not necessarily visible by looking at task-queueing delay.
- Similarly, there is evidence that renderer Impl thread responsiveness problems are caused by Chrome and can be fixed with improved scheduling.
- After Viz is implemented, the VizCompositorThread in the GPU process will always be a part of the display critical path.

Measuring these other sources of jank will increase the accuracy of our critical path estimate, at the cost of additional complexity [requires IPC, measuring a different type of latency]. For now, we choose the less complex solution. We may wish to revisit this in the future.

Appendix A: Reasons for Unresponsive Thread

There are four common reasons why a thread might be unresponsive:
- Single long running task.
- Many back-to-back queued tasks.
- CPU contention [from chrome threads/processes, from other processes].
- Significant memory pressure [blocks on I/O contention if swapping to disk, CPU contention if compressing]

Appendix B: Alternative UMA Metrics

Measuring MeanTimeBetweenJank

MeanTimeBetweenJank = (300 seconds) / (# of janky intervals). If (# of janky intervals) == 0, then MeanTimeBetweenJank = Infinity

There are two concerns with this metric

- Dealing with the 0 janky intervals case is slightly unwieldy
- Multiple reviewers commented that this was harder to grok than simply considering # of janky intervals per unit time.

Measuring FractionOfTimeSpentJanking

FractionOfTimeSpentJanking is the inverse of MeanTimeBetweenJank. It has the benefit that it doesn't have to deal with division by 0 and infinity. My concern is that it makes all the numbers very small, and thus easy to discount.

e.g., if there is 1 janky intervals in 10 seconds, then FractionOfTimeSpentJanking of 0.01 makes it sounds like not a problem, but MeanTimeBetweenJank of 10 seconds is a real problem that should be improved.