Improved expensive task scheduling policy

skyostil@, alexclarke@
January 11th, 2016
Tracking bug: crbug.com/574343

This document proposes an adjustment to the Blink scheduler's logic for blocking loading and timer tasks that it deems expensive. While blocking expensive tasks has dramatically reduced user perceived latency, it has also caused breakage in use cases where the blocked tasks are essential to the functionality of the web page.

Previous blocking logic

We block expensive timer and loading tasks in the following different cases:

1. When anticipating compositor driven gestures.
2. When anticipating main thread driven gestures.
3. During main thread driven gestures (e.g., touch and mouse drag handlers)
4. During main thread and compositor synchronized gestures.

A *compositor driven gesture* is where input events for the main part of the gesture (e.g., scrolling, pinching) are processed by on the compositor thread, bypassing Blink's main thread.

A *main thread driven gesture* is where input events flow via Blink's main thread and the compositor is not processing the gesture directly. An example is a Javascript `touchmove` handler which lets the user draw on a canvas.

A *synchronized main and compositor thread gestures* is where input events are primarily processed by the compositor, but also consumed by the main thread on a best effort basis. An example is a Javascript `scroll` listener that is running in response to compositor driven scrolling.

How we decided tasks were expensive:

1. The TaskCostEstimator is used to estimate the duration of loading and timer task queues.
    a. For loading tasks we return the 98th percentile duration of the last 1000 loading tasks. Loading tasks durations tend to follow a power law with a great number of very cheap tasks and a handful of very expensive tasks.
    b. For timer tasks we return the 90th percentile of the last 200 timer tasks.
2. The IdleTimeEstimator is used to work estimate how much time there is available to run non-compositor tasks while still hitting 60fps. It does this by subtracting the median run time of the last 10 frames worth of compositor tasks from the cc::BeginFrameArgs interval.

3. If loading or timer tasks are longer than the estimated idle time then they are deemed to be expensive because we assume running them will lead to a jank. This isn't true in all cases.

Proposed blocking logic

We proposed to only block expensive timer and loading tasks in two different cases:

1. When anticipating compositor driven gestures.
2. During main thread and compositor synchronized gestures.

In other words, we should avoid blocking any tasks during main thread driven gestures because we can't reliably tell which tasks are essential to the use case.

Additionally we propose to use different thresholds to decide a task is expensive based on the following cases:

1. When anticipating compositor driven gestures: 50ms - chosen to help us meet the RAIL response metric.
2. During main thread and compositor synchronized gestures: The estimated idle time (see above)

Bug analysis

The following bugs are related to breakage caused by blocking of heavy tasks during user input. The proposed policy would fix all of them.

MobileUTC

- Long timer delays AFTER finishing a touch scroll
    - http://flux-react-example-test.herokuapp.com (synchronised gesture)
    - Page has a touch handler which pans between different tabs horizontally. When the user stops touching, a number of expensive (~40ms) timers run to update the active tab header. Because we think that we are in a compositor driven gesture and see that timers are expensive, we block the timer code until the gesture predictor says that there is no upcoming gesture. This causes a delay of about two seconds before the tab header updates.
    - The reason why we incorrectly think this is a compositor driven gesture is because we see a ScrollBegin message in the middle of the touch event stream. Looks like this will happen when a touchstart handler does not call preventDefault().
    - We see a similar event stream for compositor gestures where there is a passive touchstart/move listener, except that in this case we also see the GestureScrollUpdates.
- Chrome mobile > touchmove > setTimeout (timeout function stops to fire while moving finger)
    - http://jsfiddle.net/juwagn/2fxe4b4f/embedded/result/ (mainthread gesture)

    - Page has an expensive rAF callback and a touchmove handler which schedules a 50 ms callback. The callback does not fire because there's no idle time left.
- setTimeout while touch in progress does not fire as expected
    - http://3dthis.com/bugs/touchbug.htm   (mainthread gesture)
    - Page has a touch handler which sets a 100 ms timer to refresh the dragged rectangle image. There is no active animation, but the first BeginMainFrame took 60+ ms, so we think there's no idle time left and the timer doesn't run.

Desktop

- setInterval and setTimeout stop during input when timer callbacks are expensive
    - https://jsfiddle.net/protozoo/x0eqntyf/4/ (pure compositor)
        - Page has an expensive animation which is implemented as a setInterval(30Hz). The animation stops while the user is moves the mouse while holding the left mouse button down.
        - This should probably be using either rAF or rIC.
    - http://live.yworks.com/demobrowser/index.html#Mindmaps
        - Cannot seem to repro even by forcing task blocking to be enabled. According to the description a setInterval based graph rendering update stops running while the user is dragging with the mouse.
    - http://www.screentoys.com/
        - setInterval-based animation stops while the user is dragging the mouse. Root cause seems to be that EaselJS defaults to not using rAF unless configured to do so.
    - http://spiral.rosslearningsystem.org/spiral/#/
        - Has since been fixed with a workaround, but similar to above cases.
- Poor performance since v47 and mouse drag
    - https://dl.dropboxusercontent.com/u/35103024/mx/test2.buildv85d/main.html
        - Uses a sequence of setTimeouts to emulate rAF. Stops running during mouse drag.

Interventions process

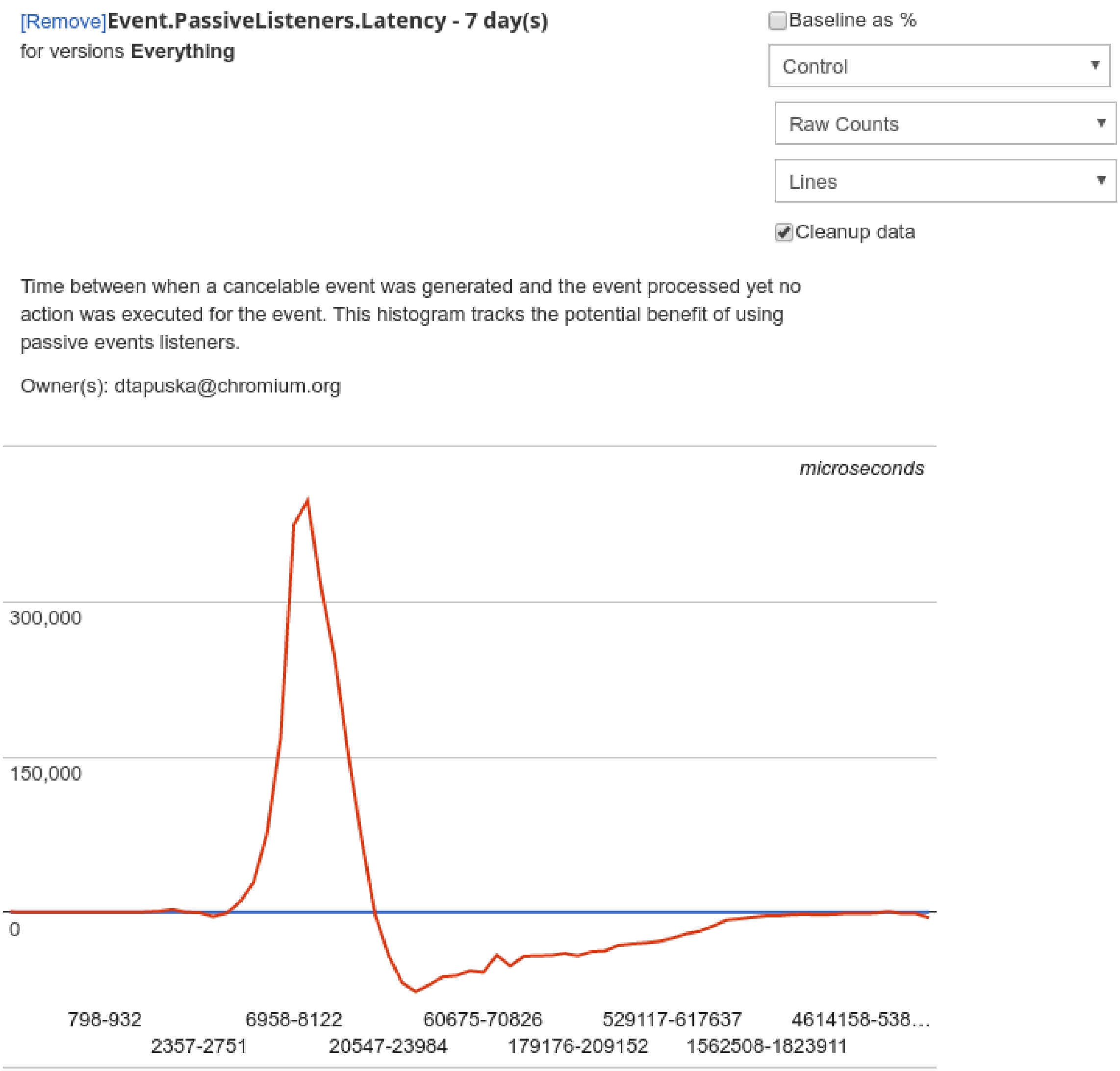
Process description:
https://docs.google.com/document/d/1E16tmoOveGopys9Fx4Nn8x8P9q7Jm6CJIqoJyz31tlg/edit

- **articulate:** Expensive task intervention policy is described above.
- **specify:** requestAnimationFrame spec has been updated to allow throttling.
- **guide:**
- Safetyhttps://www.google.com/url?q=https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail?hl%3Den&sa=D&ust=1465589584367000&usg=AFQjCNE7TtXHGKYrnjPEaeUql-CgcAOPGg

    - Delaying loading tasks, timer tasks in the 2s window before there might be a gesture
    - Only for compositor driven gestures
    - Short tasks aren't blocked
    - requestAnimationFrame and requestIdleCallback aren't blocked
    - Potential breakage
        - Timer-based animations might stop (should use rAF instead)
        - Scroll listener based animations might lag behind (this is already the case with threaded scrolling)
        - Incoming network responses might be delayed
        - Not an issue: Sending mouse motion data to a server (won't be a compositor driven gesture)
- Metrics
    - Perf bots, add a more focused first gesture latency metric
    - Measure only during loading?
    - Could do a Finch trial, but not sure if it ihope s necessary

Results from UMA

We first tested the new policy using a Finch trial in M51. Initially the improvements from performance waterfall benchmarks weren't translating to the real world, but after some investigation we found and corrected issues with the throttling and use case detection logic. After these fixes, UMA from M52 showed the median delay from passive event listeners (i.e., ones that didn't ultimately block scrolling) dropping from 50ms to 30ms (internal link): http://go/qhyxo



[Remove]**Event.PassiveListeners.Latency - 7 day(s)**
for versions **Everything**

□ Baseline as %

Control ▾

Raw Counts ▾

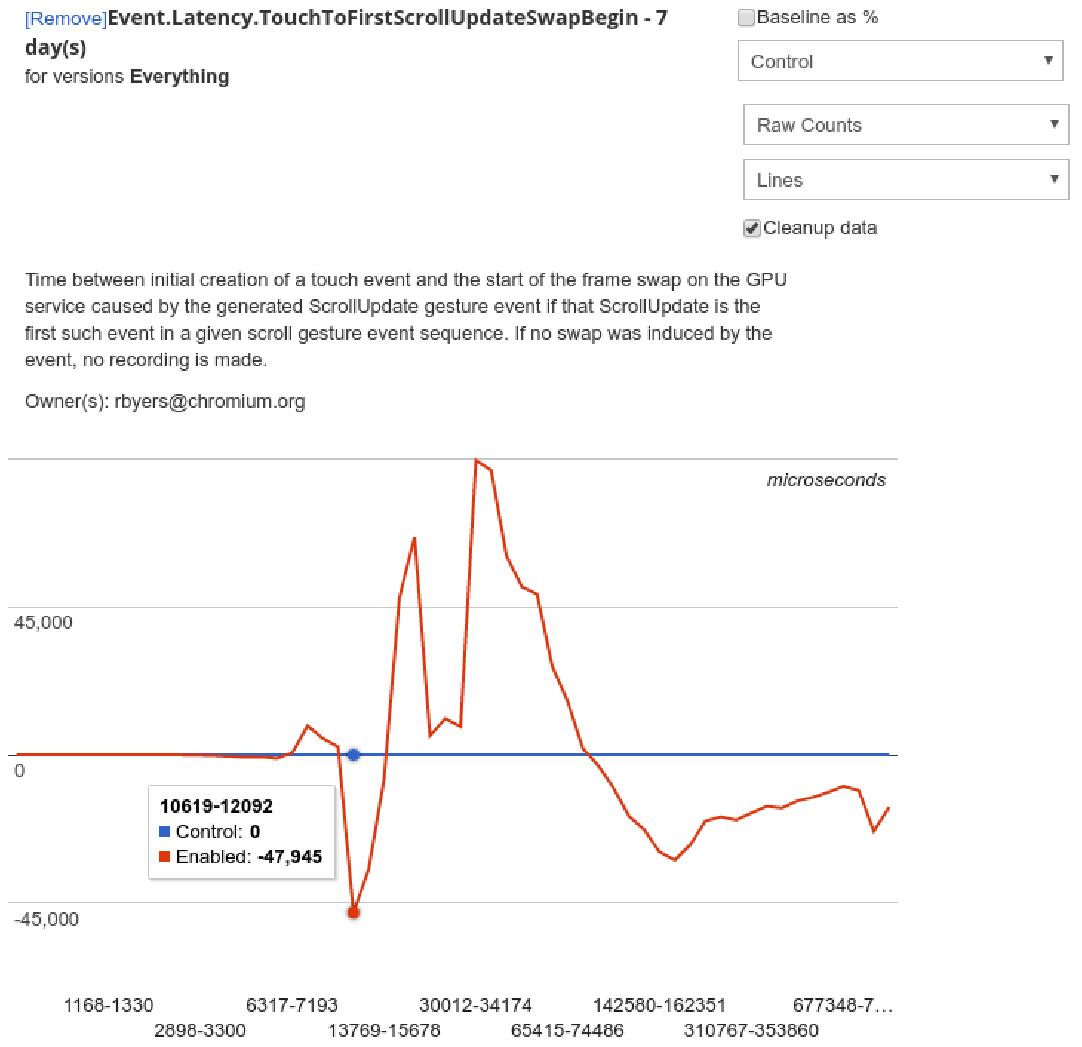
Lines ▾

☑ Cleanup data

Time between when a cancelable event was generated and the event processed yet no action was executed for the event. This histogram tracks the potential benefit of using passive events listeners.

Owner(s): dtapuska@chromium.org

The overall touch latency distribution is also showing a reduction in delays above ~70ms:



[Remove]**Event.Latency.TouchToFirstScrollUpdateSwapBegin - 7 day(s)**
for versions **Everything**

☐ Baseline as %

Control ▼

Raw Counts ▼

Lines ▼

☑ Cleanup data

Time between initial creation of a touch event and the start of the frame swap on the GPU service caused by the generated ScrollUpdate gesture event if that ScrollUpdate is the first such event in a given scroll gesture event sequence. If no swap was induced by the event, no recording is made.

Owner(s): rbyers@chromium.org

https://codereview.chromium.org/15700.h"   8 #include
"components/renderer_context_menu/context_menu_delegate.h"
 9.h"   8 #include "components/renderer_context_menu/context_menu_delegate.h"
 983003/.h"   8 #include "components/renderer_context_menu/context_menu_delegate.h"
 9

Based on this we decided to ship the intervention in M52.

Patches

- Fix use case detection with mixed touch and scroll events: