ZilCh (Zero Input Latency Chrome)

- Tracking bug: https://code.google.com/p/chromium/issues/detail?id=168459

Note: This document is meant to serve as a high-level background and plan for potential latency improvements. Other documents, which are linked to at the end of this one, will include more details on specific types of optimizations and platform-specific details.

There has been a lot of work to improve Chrome's latency, but there is still a lot left to do. Chrome's end-to-end latency is currently worse than the competition. In order to decrease our latency as much as possible, we need to consider all stages of the pipeline from Chrome itself to the hardware and drivers. With the right combination of optimizations and features, sub-vsync latency should be possible in many cases.

Rounding down to the nearest vsync, we are calling this effort Zero Input Latency Chrome (ZILCh). Alternative names: Project Zero Latency (PZL), Zero Latency Project (ZLP), Chrome Zero Latency (CZL), Zero Latency Display (ZLD)? Suggestions welcome!

ZILCh will require us to tackle every part of the critical path. In some cases, it may make sense to insert shortcuts from touch to display. Eventually, we may even have shortcuts that are set up entirely in hardware and avoid the CPU all together!

Before we get too far ahead of ourselves, however, there are easier inefficiencies to tackle. The short-term focus will be on implementing high-level software optimizations to our critical paths that benefit as many platforms as possible.

Goals: Not just zero latency

Our obvious goal is to http://build.chromium.org/p/chromium.webkit/builders/GPU%20Win7%20(dbg)%20(NVIDIA)have the lowest latency possible from user input to the user's eye, but we have to be careful not to sacrifice our other goals in the process. We list the metrics we plan to optimize here to provide a sense for what this effort considers important and also to guide our design decisions. Eventually, there will be tradeoffs between these metrics and we will need to determine the combination of metric values we want to target that provide the best user experience.

Metrics to optimize:

TODO: Turn these into actual equations.

1. Latency of initial user input events, e.g. touch-down.
2. Latency of continuous user input, e.g. dragging.
3. Completeness of frames rendered.
4. Variance in latency, i.e. latency jank.

5. Throughput.
6. Synchronization between content.

Testing

Notable test cases are listed here to provide a sense of all the different code paths and scenarios we will need test. They are categorized by the starting point of the latency we want to consider. Each starting point includes a list of dimensions that can be mixed and matched to create different scenarios. It will be impossible to test all combinations, so we will need to determine which scenarios are important to us and will provide the best overall coverage.

Test cases:

1. Main Renderer thread latency:
    - a. Dimension 1: Main-thread scheduling method.
        - i. RequestAnimationFrame.
        - ii. Timer-based thread scheduling (e.g. setInterval()).
        - iii. User input driven frames.
    - b. Dimension 2: Type of user input.
        - i. Initial user input (e.g. tap/click).
        - ii. Continuous user input (e.g. drag) throttled by vsync in OS. (Android)
        - iii. Continuous user input (e.g. drag) not throttled by vsync in OS.
    - c. Dimension 3: Types of content producers
        - i. DOM manipulation requiring layout and repaint.
        - ii. DOM manipulation only requiring repaint.
        - iii. Canvas (Software and hardware paths)
        - iv. WebGL.
        - v. Video.
        - vi. Pepper (Flash/NaCl)
    - d. Dimension 4: Potential throughput bottlenecks.
        - i. No bottlenecks.
        - ii. Expensive input handler.
        - iii. Expensive layout.
        - iv. Expensive painting.
        - v. Expensive image decode.
        - vi. Expensive compositor quad generation.
        - vii. Expensive compositor quad drawing on GPU.
        - viii. Expensive texture uploading.
    - e. Dimension 5: Potential sources of jank.
        - i. No sources of jank.
        - ii. Deschedule Main thread for M milliseconds every N frames.
        - iii. Deschedule Compositor thread for M milliseconds every N frames.
        - iv. Deschedule Browser thread for M milliseconds every N frames.
        - v. Deschedule GPU thread for M milliseconds every N frames.
        - vi. Deschedule Texture upload thread for M milliseconds every N frames.

        vii. Draw an expensive frame for GPU hardware every N frames.
    f. Dimension 6: Interaction between multiple producers.
        i. User input.
        ii. RequestAnimationFrame.
        iii. Compositor animations.
        iv. Browser animations.
    g. Dimension 7: Shortcut paths. Setting up shortcut paths in the main thread will require web extensions that should be tested eventually, but are not a priority.
        i. No shortcuts used.
        ii. Shortcuts to Compositor thread.
        iii. Shortcuts to the Browser thread.
        iv. Shortcuts to the GPU thread.
    h. Dimension 8: Process/thread modes
        i. Fully separate.
        ii. Merged threads. (WebViewUI)
    i. Dimension 9: Rendering modes
        i. Software compositor
        ii. Hardware compositor
    j. Dimension 10: Readbacks
        i. No readbacks
        ii. With WebGL readbacks
        iii. With tab capture / mirroring
    k. Dimension 11: Notable Corner Cases
        i. The compositor has to redraw a frame because part of the previous frame was low-resolution or incomplete/checkerboard.
2. Compositor thread latency
    a. Dimension 1: Use case
        i. Accelerated touch panning.
        ii. Accelerated pinch zooming.
        iii. Accelerated touch fling start and stop.
    b. Dimension 2: Type of main-thread touch handler.
        i. Without javascript handler. No round trip to the Blink thread required.
        ii. With fast (<< 1 vsync) reject javascript handler.
        iii. With slow (> 1 vsync) reject javascript handler.
        iv. With timed-out javascript handler. Timing out the javascript handler if it takes too long would break the web platform. It's unlikely we will implement it, but keeping it here for completeness.
    c. Other dimensions that apply from main thread latency.
3. Browser thread latency
    a. Dimension 1: Use case
        i. Aura
            1. Window move.
            2. Tab switch.

            3. Window resize. This is a special case because the Browser has to wait for new content of the correct size from the Renderer.
        ii. Android
            1. Single tab shown.
            2. Side swipe between two tabs.
            3. Fling through tab stack.
    b. Dimension 2: Content generation speed. If the Browser requests a new frame from the Renderer, like during window resize, we will need to take into account the round-trip time to the Renderer.
        i. With fast Renderer frames.
        ii. With slow Renderer frames.
    c. Other dimensions that apply from main thread latency.

Previous and Current Work

There has been work recently to measure Chrome's latency and to lower Chrome's latency. Although not comprehensive, this section should provide some context into recent latency work.

Measuring Chrome's Latency

We need a way to measure our progress. The following efforts allow us to do so:

1. An automated way to measure chrome's software latency. This will allow us to catch obvious latency regressions. (jbauman/miletus) https://docs.google.com/a/google.com/document/d/1UEoVttE9UMNs3ouL0tZ41GAMkOk732dNIaP73gBWYd4/edit
2. Buttermark latency benchmark. Emulates user input and captures the screen. An all software solution that captures more of the pipeline. (jdarpinian) https://www.corp.google.com/~jdarpinian/buttermark.html
3. Total end-to-end latency comparison using high speed cameras. This gives us a bigger picture that includes total latency from finger position to pixel output. (flackr) https://docs.google.com/a/google.com/document/d/1MU92iEOdoYcrqnSx1EI-UKa9J7TordEQQ6KqPxzsBp4/edit
4. Quantification of how much latency comes from kernel/touch controller vs UI/GPU processing on Android: Deschedule https://docs.google.com/a/google.com/presentation/d/1eMqE2Fiw0SL-ar-SDLUS-bBBoqpaosafc1I0yPG5Uy4/edit#slide=id.p

Relevant bugs and patches

This list is incomplete, please help fill it in.

Bugs:

- Improving latency on Android (master bug for Android): https://code.google.com/p/chromium/issues/detail?id=230759
- Prioritize touch event handling on WebKit message loop: https://code.google.com/p/chromium/issues/detail?id=173543

- Noticeable latency on canvas2d paint app: https://code.google.com/p/chromium/issues/detail?id=99171
- Commit-less update of renderer contents in browser compositor https://code.google.com/p/chromium/issues/detail?id=229742

Patches:

- android: Remove touch slop analytically instead of ignoring first scroll: https://chromiumcodereview.appspot.com/12457038
- Plumb cc::LatencyInfo through command buffer and output surface: https://codereview.chromium.org/12614013/
- Send vsync notification from browser to renderer https://codereview.chromium.org/13068002/
- Implement vsync notification on Android https://codereview.chromium.org/1195903
- Provide vsync signal to compositor https://gerrit-int.chromium.org/#/c/30041/
- Include timestamp with vsync-delineating input events https://codereview.chromium.org/13880009/
- cc: Use input events to trigger vsync https://codereview.chromium.org/13863006/
- Android: Use input events to trigger vsync https://codereview.chromium.org/13947036/

Background

This section serves to provide information about things we should keep in mind when trying to lower latency, including the need to address jank and inherent shortcomings of shortcut methods.

A quick introduction to jank

Jank is a temporal visual disturbance that interrupts a sequence that would otherwise be smooth. It is very important to understand jank so that our latency mitigation strategies do not introduce jank. There are actually two types of jank:

- **Throughput jank:** This is the most common and obvious type of jank. It shows up when a frame takes more than 1 vsync to produce, causing us to skip a frame.
- **Latency jank:** Even if we are producing a frame every vsync, it is still possible for the output to look rough if latency from any producer to the display is not constant. For example, latency jank can crop up if we do not have a consistent animation time source or if we process input in variable intervals. The effect of latency jank can be small, but is perceivable enough to be annoying.

It is interesting to note that most throughput janks are accompanied by two latency janks:

- The first latency jank occurs when the missed frame hits the display in L+1 vsyncs when it was supposed to hit the display in L vsyncs.
- The second latency jank occurs when the application jumps back to L vsyncs of latency from L+1 vsyncs of latency. Whether this jump should occur in a single frame or should be spread out over multiple frames is TBD. See the “latency recovery strategy” action item.

Potential issues with shortcut paths

Shortcut paths are good for deferring work to run later in the display pipeline, but are not a panacea. The later pipeline stages will not have as much context to work with as the earlier pipeline stages and the amount of work we can delay must be limited. If the delayed work takes a long time, we lose the latency benefits of deferring the work.

These limitations can cause problems, from unavailable content to synchronization issues.

Not having content to display later in the pipeline

Let's consider a simple shortcut path injected right before hardware overlay positions are determined. One use case this shortcut path enables is panning of a single layer. The problem is that previously out-of-bounds content must be available to scroll on to the screen. Similar problems will crop up with any shortcut path and, while it may be feasible to perfectly handle small adjustments in a shortcut path, it may be unreasonable to handle large adjustments well. Mitigations strategies must be implemented on a case by case basis.

Lack of access to program state later in the pipeline

We will likely prevent a shortcut from communicating back to an earlier pipeline stage in order to keep things simple. Preventing backwards communication could cause the display to be out of sync with the program's state. The program will get the same input events as well, so everything will eventually correct itself, but cases to watch out for are non-linear events. For example, if we want to stop panning when we reach the end of a layer, it would be bad for the shortcut to pan past the end of that layer.

Synchronization issues

There are potential synchronization issues with shortcut paths:

1. **Multiple producers:** Consider two producers of content, for example WebGL with a Canvas overlay. If synchronization between the two producers is required, both producers must implement shortcut paths that take the same amount of future input into account.
2. **Stacked shortcuts:** Now suppose we have a single producer of content, but we want to apply two shortcut paths at different parts of the pipeline. For example, the first shortcut could feed into the GPU thread and the second shortcut could feed into an overlay position. If we always applied the input delta from the time or production, the first part of the input delta would be applied twice. We will need a way to communicate and subtract the overlapping delta from the second shortcut.

Inspiration from the Carry-Select Adder

In a ripple-carry adder (http://en.wikipedia.org/wiki/Adder_(electronics)#Ripple-carry_adder), the carry chain is the critical path. One of the simplest optimizations is the carry-select adder (http://en.wikipedia.org/wiki/Carry-select_adder), which shortens the critical path by calculating the two possible outcomes a) if the carry-in is 0 and b) if the carry-in is 1. Once the actual value of the carry-in is known, the correct answer of the two possible outcomes is selected.

Although the carry-select adder optimization shortens the critical path, it does so at the cost of wasted work. With this tradeoff in mind, we can perform similar optimizations in our critical path if we have spare CPU to burn. For example, we can hide texture upload completion latency by producing one frame for a) if the uploads have completed and b) if the uploads haven't completed; selecting the best frame to use as late as possible.

Can we use this concept anywhere else?

Additional Background

Consider >60Hz displays

https://docs.google.com/document/d/1zS0RNqppwszfhed7Dp_JSKGmnvLKlRbaWQKcYJG11wA/edit?usp=sharing

This document discusses issues with trying to control latency on high frequency displays, especially if you wish to target a specific latency.

Carmack article detailing how to achieve 2-3ms latencies with a VR headset.

http://www.altdevblogaday.com/2013/02/22/latency-mitigation-strategies/

Actionable Items

Listed here are ideas that need investigation and/or implementation. They are listed roughly in order of priority based on dependencies, potential latency gains, and complexity. Even though some of the medium and long term goals aren't necessarily implementable today, we still outline them here so we can take the proper steps now to unblock their implementation in the future. Some of the reach goals, in particular, would require coordination with hardware vendors or extensions to the web.

High Level Goals

1. Automated testing framework with directed tests.
2. Schedule our pipeline stages better (Main|Compositor|Browser|GPU|Hardware).
3. Find and remove places where the pipeline blocks unnecessarily.
4. Add shortcut paths from user input to as late in the pipeline as possible.
5. Move as much work off the main Renderer thread as possible.

Short Term

1. Determine how exactly we will calculate the metrics we care about in an automated way on the bots.
2. Write directed tests for use cases listed in the Testing section that provide good coverage.
3. Analyze more traces of bad and common use cases to identify bottlenecks.
4. Right alignment to VSync for user input, RequestAnimationFrame, Renderer compositor, and Browser compositor.

    a. For a detailed description, refer to: https://docs.google.com/a/chromium.org/document/d/1LUFA8MDpJcDHE0_L2EHvrcwqOMJhzl5dqb0AlBSqHOY
    b. The idea here is that we can schedule our stages more intelligently by compressing them as close to the display's vsync as we can. Generally, all our stages are currently clocked by the same vsync.
    c. As part of right alignment, we might want to trigger a pipeline stage when all inputs are ready OR on a deadline if only a subset of inputs are ready.
        i. Triggering when all inputs are ready gets the stage started as soon as possible. If all inputs are ready, there is no reason to wait.
        ii. Triggering on a deadline, if only a subset of the inputs are available, allows some inputs to take effect without stalling everyone.

5. Find ways to address latency introduced by blocking commits.
6. Determine how we should handle long-running javascript input handlers.
    a. Frame-select draw. (See carry-select adder)
        i. Make frame assuming javascript has rejected user input.
        ii. If actually rejected, we are ahead of the game, otherwise start over.
        iii. Good for quick rejects.
    b. Timeout
        i. Assume javascript will reject the input event after some timeout.
        ii. Recover if javascript actually handled the event.
        iii. Good for long rejects, questionable if input event is actually handled.
7. Address GPU Thread latency. When the GPU thread is busy, it can get backlogged.
    a. Make our GPU service faster.
        i. Push driver vendors to optimize their drivers so they don't block unnecessarily.
        ii. Optimize any overhead in the GPU service code itself.
    b. Adding a shortcut path for simple use cases. For example, modify geometry in the GPU thread based on user input that the Compositor thread has claimed.
        i. What are the security implications here?
        ii. Short term: Implement extension to set mouse/input position uniforms in GPU thread to be used by vertex/fragment shaders.
        iii. Longer term: There are vertex and fragment shaders; why not input shaders to remove redundant work from vertex shaders?
8. Improve round-trip upload completion latency and synchronization.
    a. Get rid of 6ms raster/upload check delay (kCheckForCompletedTasksDelayMs).
    b. Skip the regular GPU pipeline for async uploads.
    c. Zero copy: Just draw into a memory-mapped buffer and use it.
        i. Avoids 1st trip of the round-trip upload completion latency.
        ii. Platform implementation dependent.
        iii. Must beware of hidden cache flush hits on software/hardware lock handoffs though. Depending on the platform, it may be beneficial to delay and group buffer unlocks.

    d. Texture-select bind. (See carry-select adder)
        i. Would defer texture selection to occur on GPU thread based on uploads that have completed.
        ii. Avoids the 2nd trip of the round-trip upload completion latency.
        iii. Limitation: cannot handle layer topology changes. See Übercompositor's frame-select draw.

9. Canvas and WebGL commits block the main thread waiting for pending tree activation.
10. Prioritize input events ahead of others. This will be tricky, but putting input events in a separate queue will prevent them from getting stuck behind a slow task.
    a. Is it possible to forward input events through I/O threads until they reach the correct destination?
11. Determine our latency recovery strategy: when our latency is high because we missed a vsync, how do we want to transition back to a lower latency?
    a. Does it feel better to jump the latency down a whole vsync all at once or does it feel better to gradually step the latency down fractions of a vsync over multiple frames? Gradually catching up would result in the display looking like it is running slightly faster during the catch up period.
    b. Would it make sense to have different strategies for animations and user-input?
    c. At what point should we target a lower framerate to reduce jank?
12. Determine our latency policy.
    a. For some apps (rhythm games), a consistent latency may be more important than lowest-latency.
    b. In other cases (first person shooters), you may just want the lowest latency you can get.
    c. Should we support multiple latency policies?
13. Übercompositor
    a. Throwing away renderer frames becomes much cheaper because they aren't actually drawn on the GPU; you only waste CPU setup time.
        i. Frame-select draw. (See carry-select adder.)
            1. Select one of two frames as late as possible based on texture upload completion.
            2. Different than texture-select bind in that layer topology can also change.
            3. Limitation: likely more expensive than texture-select draw.
        ii. Do not throttle the Compositor thread, let the Browser thread discard old frames and just use the newest.
    b. Remove commit blocking flow when there is already 1 commit pending. Allow N queued commits and use end-to-end credit-based flow control instead.
    c. Improve resizing latency. This is a special case where it will be better for the Browser to wait (with a timeout) for the Renderer frame with the correct size rather than compositing with the wrong size.
14. Interpolate user input on non-Android platforms.
    a. User input will look the smoothest if it is synchronized to vsync.

Medium Term

1. Move as much work off the main thread as possible. Freeing up the main thread will allow it to handle or reject input events more quickly. See work being done by the Chrome Speed team.
    a. Threaded styling: https://docs.google.com/a/google.com/document/d/1dyIlLudVxZiFEmE_fIoAMwvcl-Edaxxi2EXjg_ECIDU/edit#heading=h.vle3fiohy52f
    b. Threaded layout.
2. Allow web developers to opt-in to a mode where touch scrolling never blocks on main thread input handlers
    a. touch-action from W3C PointerEvents can accomplish this
    b. https://code.google.com/p/chromium/issues/detail?id=258459
    c. https://docs.google.com/a/chromium.org/document/d/1CV2AXyrdPdGSRypAQcfGrgQVuWYi50EzTmVsMLWgRPM/edit?usp=drive_web
3. Get feedback from the windowing system regarding frame queue depth. Without this information, Chrome cannot know the latency of the eglSwapBuffers call.
    a. Completion time using sync points: https://codereview.chromium.org/14358014/
    b. Android planning to add an extension for a frame completion/display time.
        i. See eglPresentationTimeANDROID and future extensions.
    c. Supported by Windows via DWM API's, but we can't use them because of a Windows bug when also using some GDI functions. We should move away from GDI functions if we can.
    d. We need something similar on other platforms (Linux, ChromeOS).
4. Provide tools to web developers that flag poor latency cases / scenarios.
5. Use hardware overlays, then add shortcut path from touch to overlay size/position.
6. Prediction of input.
    a. Controversial, but small amounts (< 1 vsync) of prediction might be acceptable.
    b. Once our latency has improved in other places, a small amount of prediction will be relatively more significant.

Long Term / Tentative / Pie in the Sky

1. Driver optimizations.
    a. Finer control of GLES context priority and pre-emption.
    b. Shortcut paths in drivers.
    c. Frame-select optimizations
        i. On tiling architectures, for example, we could set up the parameter buffers for two sets of tiles, but only rasterize one of them.
2. Hardware optimizations.
    a. Two logically concurrent GPU's might provide better scheduling guarantees, so we don't have to time multiplex our graphics contexts. Each GPU could be optimized for particular workloads:
        i. Large numbers of triangles and throughput (Painting/WebGL).

        ii. Fill rate and low latency (Compositing).
            1. Ideally, tiles can be rendered just-in-time for scanout if we meet certain restrictions: overdraw, number of texture sources, vertex/fragment shader complexity, etc. Saves writing to a temp buffer.
            2. If it can’t be done during scanout, we would risk underflowing the display and should fall back to a double buffered mode.
    b. Shortcut path from touch firmware to GPU hardware.
3. 120Hz/240Hz display.
    a. Being able to queue multiple commits/frames and have fine control over the latency becomes very important in order to better parallelize the pipeline.
    b. For an EGL extension proposal for latency management, see: https://docs.google.com/document/d/1zS0RNqppwszfhed7Dp_JSKGmnvLKlRbaWQKcYJG11wA/edit?usp=sharing
4. Possible web extensions.
    a. Shortcut HTML extensions. Much discussion is needed in this area, but the general idea is to have an asynchronous function that can turn the same knobs as a CSS animation, but dependent on future user input events. This function should be applied as late in the pipeline as possible. Fixed-type functions could be defined if that allows them to be applied later in the pipeline than javascript otherwise would be. Could something like asm.js be used here to avoid security holes of running javascript in the GPU thread?
    b. Shortcut WebGL extensions. Much discussion is needed in this area, but the general idea is to have something like an asynchronous function that can mutate uniforms later in the pipeline based on future user input.
        i. The Maps team has expressed interest in this solution.
    c. Shortcut Canvas extensions? Less clear how this might work.
5. Display interface optimizations?
    a. Store 2D overlays in the display.
    b. Allow shortcut paths from touch firmware to overlays.

Follow-up documents

TODO:

1. Testing details
2. Platform agnostic work
3. Android specific work
    a. https://docs.google.com/a/chromium.org/document/d/16822du6DLKDZ1vQVNWI3gDVYoSqCSezgEmWZ0arvkP8
4. CrOS / Übercompositor specific work
5. Windows specific work
6. Mac specific work
7. Shortcut Latency Web Extensions
8. Driver improvements and shortcuts

9. Hardware improvements and shortcuts