GPU Service Scheduling Latency

Contributors: brianderson, skyostil

Status: Public. Mostly superseded by Trustable Future Sync Points.

Tracking bug: https://code.google.com/p/chromium/issues/detail?id=350608

This a followup to Chrome: Frame Synchronization, which outlines scheduling issues we can run into in Chrome with Ubercompositor and double-buffered WebGL/Canvas2D. One of the problems is that the WebGL/Canvas2D context can be scheduled on the GPU service earlier than it is actually needed, which delays execution of more urgent UI commands. The purpose of this document is to propose a way to improve our worst-case latency without sacrificing our best-case latency by specifically targeting WebGL/Canvas2D vs UI context ordering on the GPU service.

With the combination of **shallow flush hints**, **service-side flushing of sync points,** and **weak future sync points**, we should be able to give the UI enough control of the GPU service to prevent WebGL/Canvas2D from running early and hurting overall latency. With additional optimizations, we may be able to improve best-case latency as well.

Description of the Problem

If we run WebGL commands on the GPU service too far ahead of the UI commands that consume the WebGL target, then we unnecessarily push back the UI frame's completion and when it can be displayed.

1. The following is an example of contexts being scheduled in a poor order:
    a. WebGL FB1
    b. UI FB1
    c. WebGL FB2
    d. UI FB2, which reads from WebGL FB1
2. It would have been better if the contexts were scheduled in the following order, only producing the WebGL framebuffer immediately before it is consumed:
    a. UI FB1
    b. WebGL FB1
    c. UI FB2, which reads from WebGL FB1
    d. WebGL FB2

If we try to fix (1) simply by always deferring WebGL commands until the UI consumes the frame, however, we will hurt best-case latency if the WebGL commands could have finished in time for the vsync but did not because of the deferral.

High Level Idea

Have WebGL/Canvas contexts operate in two modes: a low latency and high latency mode. To enable these modes, we will add a shallow flush hint to the GPU service, that periodically kicks the GPU service, but may or may not result in execution of commands on the GPU service. We will then add sync point features that allow the UI to select between low and high latency modes for the WebGL/Canvas contexts that determine how the GPU service should respond to the shallow flush hints.

Low Latency Mode : The Shallow Flush Hint

The shallow flush hint has a weaker guarantee than a shallow flush in that a shallow flush results in execution on the GPU service soon, but the hint might not.

1. Many existing shallow flush calls can be converted to shallow flush hints.
2. Shallow flushes should be used for cases such as glReadPixels or when the command buffer is full, where the client is waiting on the service to return soon.

In the low latency mode, we could operate as we do today, where the commands start to execute after frame submission, or we could stream commands before frame submission to the GPU service to improve best-case latency.

If we are streaming a context, the GPU service will respond to shallow flush hints and execute commands immediately. We will need to optimize the hint frequency since there will be an overhead vs. latency tradeoff.

High Latency Mode : Service-Side Flushing

In the high latency mode, we do not want the GPU service to respond immediately to the shallow flush hints. Instead, we will defer GPU commands until consumption on the UI active tree. The shallow flush of the WebGL context will be triggered by the UI context on mailbox consumption rather than mailbox submission. ,

To enable high latency mode, we need to support creating sync points that don't result in an immediate shallow flush. The shallow flush will be triggered service-side from the UI's context instead:

1. This will allow WebGL to submit a frame without necessarily causing its commands to execute immediately.
2. Waiting on the sync point will shallow flush the appropriate channel service-side, up to the sync point.
    a. A UI frame should wait on all the sync points it wishes to use before drawing its frame to avoid switching contexts too often mid-stream.
    b. We will need to store metadata regarding which channel the sync point is on so we can shallow flush the appropriate channel.

Selecting between Low and High Latency Modes : Future Sync Points

To support switching between low latency mode and high latency mode, we will add future sync points. Future sync points are created and signaled by separate functions. In comparison, normal sync points are created and signaled by the same function. Separating the creation and signaling of a sync point allows a channel to wait on a future syncpoint before it has been signaled.

Future sync points will be used in the following way:

1. UI creates a future sync point and sends it to the Renderer.
2. The Renderer inserts a service-side wait on the future sync point before queuing its commands. This wait is in addition to and comes after the existing wait on the normal syncpoint that prevents rendering to a buffer while it is still being read from.
    a. To differentiate between the two sync points, we might want to refer to the existing sync point as the “correctness sync point” and the new one as the “scheduling sync point”.
3. To support glReadPixels() and draining full command buffers, the wait on the future sync point must be canceled if there are one or more pending shallow flushes on the waiting channel.
    a. This requirement makes our future sync points “weak”.
    b. The existing wait on the “correctness sync point” ensures the cancellation does not result in a context drawing to a buffer that is still in use.
    c. Strong future sync points whose waits cannot be canceled might be useful in other situations, but are very deadlock prone. See discussion at the end of this document.
4. The UI can select between low and high latency modes for a given Renderer by changing when it signals the future sync point.
    a. Signaling immediately after the first draw = low latency mode.
    b. Signaling it on consumption of the next mailbox = high latency mode.

Security and Deadlock Concerns

Since only the UI creates future sync points, we prevent Renderers from being able to block the GPU process indefinitely.

Implementation Details

Service-Side Flushing and Low-Latency Sync Point Creation

Creating a sync point currently needs a client-side wait for an IO thread round trip to the SyncPointManager in the GPU process. Can we do better by allocating sync points client-side and verifying service-side, avoiding the client-side wait? This should be easier to implement if each GpuChannel has its own SyncPointManager and can independently increment its sync points without having to coordinate with other channels. The MSBs of a sync point can be reserved to hold the channel number. This may require representing sync points as a struct, but would have the following benefits:

1. The sync point itself encodes which channel will need a shallow flush on a wait if the syncpoint hasn't been signaled yet.
2. The sync points within a channel will be signaled in the order they were created, which will allow SyncPointManager to:
    a. store outstanding sync points in an ordered list, rather than a hash table, and
    b. determine if a sync point has been signaled or not with a overflow-protected comparision.

If the flush resulting from a wait on a sync point reaches the end of the available commands without signaling the sync point, that is an error condition. How to react TBD.

Future Sync Points

A full-blown out-of-order retirement of sync points will be overkill. If we restrict future sync points to retire in-order with respect to other future sync points on the same channel, we can satisfy our requirements, simplify our implementation, and increase the chances that future sync points may be implemented in hardware.

1. Use two SyncPointManagers per GpuChannel. One regular, one future. This allows future sync points to signal out-of-order with respect to regular sync points.
2. Simplifications and Restrictions
    a. Regular sync points will be created and owned by the channel itself, whereas the future sync points will be created and owned by a single parent channel. Supporting multiple parent channels would require more than two SyncPointManagers per channel for an unknown benefit. This will allow the UI to signal a Renderer out-of-order with respect to other Renderers.
    b. Future sync points must still be signaled in-order with respect to other future sync points within a channel.
    c. A channel may only wait on a future syncpoint existing on its own channel, which was created by its parent channel according to 2.a.
3. Reserve 1 bit on the sync point (in addition to the channel bits) indicating future status.
4. Add entry points CreateFutureSyncPoint and SignalFutureSyncPoint.
5. How to cancel a wait on a future sync point.
    a. Keep a counter of pending shallow-flushes on a channel.

    b. If counter is not 0, re-evaluate whether or not the channel should be scheduled.
6. Security and Deadlock Concerns
    a. Allowing only one parent to create future sync points for a given channel avoids degenerate circular dependencies if the “root” context has no parents.
    b. How should we handle loss of the UI (parent) context?
    c. How should we handle loss of the WebGL (child) context?

Discussion

Weak vs. Strong Future Sync Points

The future sync point described in this document so far is a “weak” sync point, in that a shallow flush will cancel waits on it. We may want to consider implementing a strong future sync point that does not cancel waits on a shallow flush. This can be used, for example, in the following two ways:

1. To release a mailbox to WebGL client-side without necessarily having released it service-side yet, which will enable additional main-thread concurrency without triple buffering. (i.e.: use a future sync point for the “correctness sync point”).
2. Allow the Renderer to submit its future frame contents to the UI before the content is actually finished.

Strong future sync points are much more deadlock prone, however. For example:

1. NPAPI requires us to complete a commit once it has started, but having a wait on the strong future sync point could prevent a shallow flush from finishing during the commit because it is waiting on the Browser process to release a mailbox, but the Browser Process is busy handling the NPAPI plugin and will never get to release the mailbox.
2. Having the Browser’s context block indirectly on the Renderer’s CPU runtime might introduce unwanted jank.

Other Applications for Future Sync Points

1. Conditional Texture Bind.
    a. See entry in the ZilCh doc.
    b. Run texture upload in a completely parallel context, signaling a future sync point when the textures required for activation have been uploaded.
    c. Add method to test if the future syncpoint has been reached without waiting.
        i. Client-side test would be straightforward.
        ii. Service-side test would require API for storing test result in a variable to ensure atomicity of the decision.
            1. Could we implement the variable as a syncpoint in a dummy context?
    d. Add function to bind a texture whose id depends on the result of the test.
2. Frame-Select Optimizations
    a. Triggered just like Conditional Texture Bind, but draws different geometry rather than just binding different textures.