Table of Contents

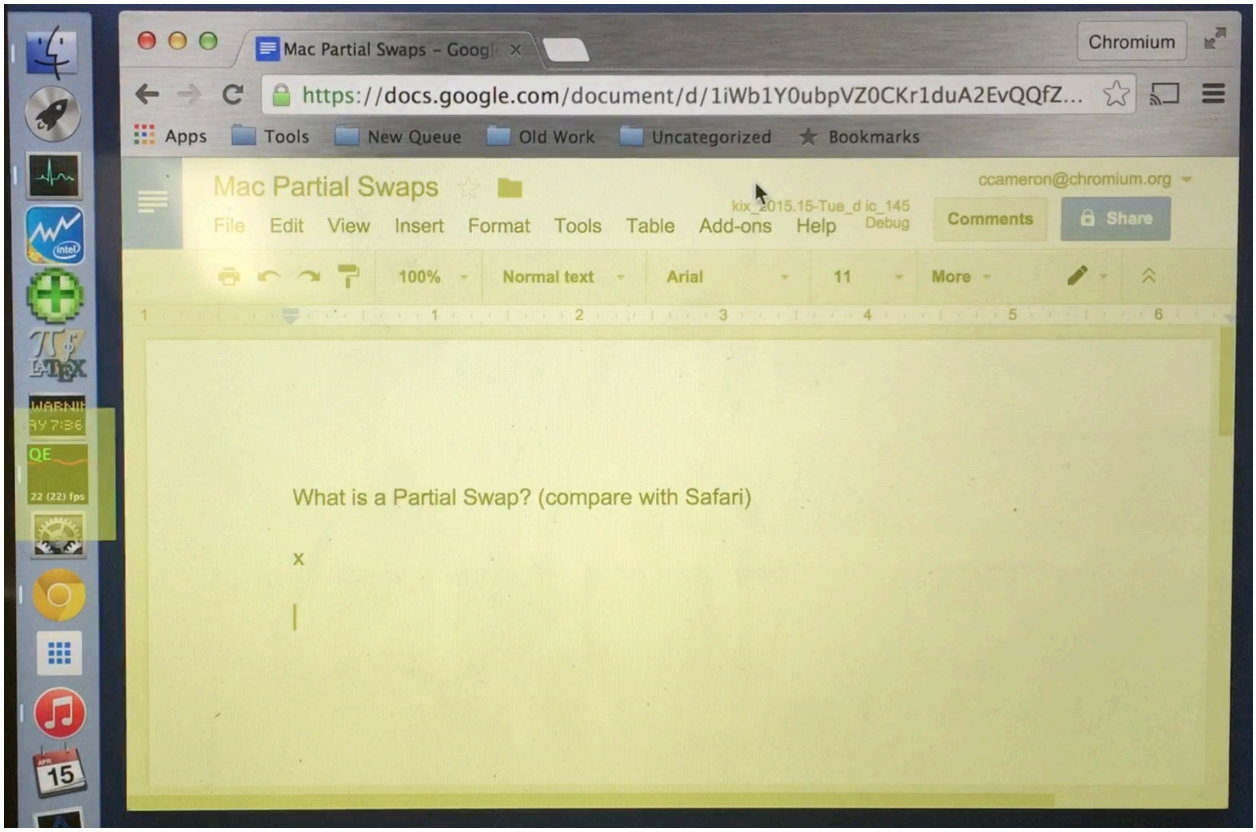
ccameron@, 2015-04-15 (ish)

What is a Partial Swap?

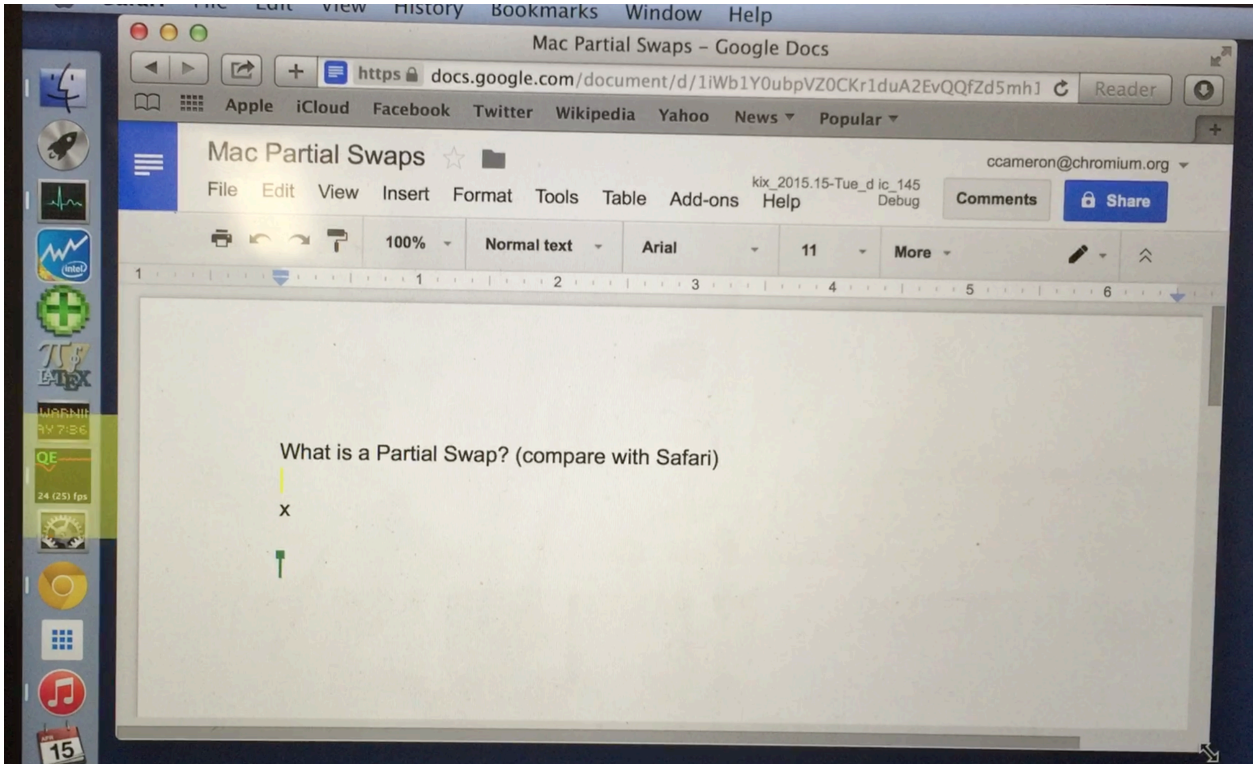
Partial swapping refers to only drawing the part of the screen that actually changed. The part of the screen that actually changed is often referred to as a “damage rectangle” or “damage rect”.

On Mac, we can see which part of the screen changed by using QuartzDebug, and selecting “flash screen updates”. This will make parts of the screen that change flash in yellow.

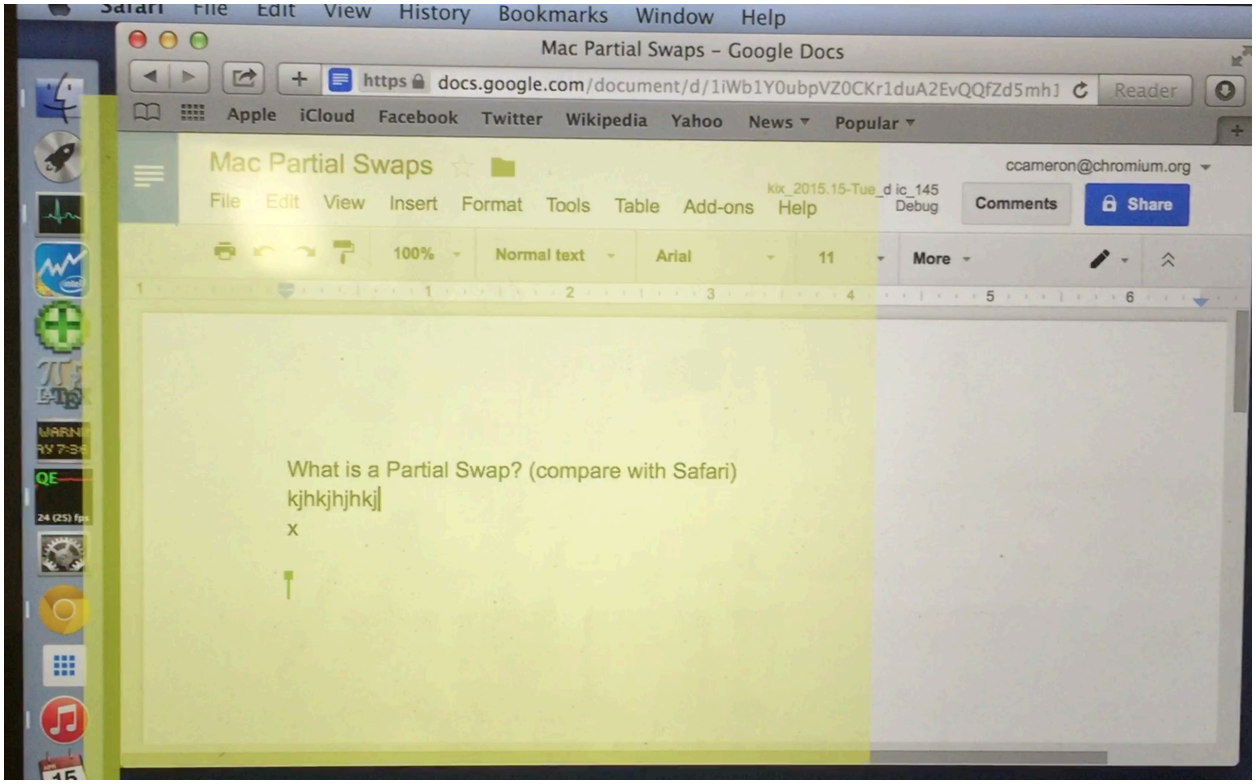
If you try that with this document in Chrome, you will notice that the blinking cursor causes the entire content area of the screen to change (see the attached screenshot … note that cell phone videos are pretty much the only way capture this).



If you try the same experiment with Safari, you'll notice that the only part of Safari's screen that updates is the actual cursor (it's hard to see in the screenshot below, but it's under the "W" in "What".



Safari tracks damage in large tiles. In the below screenshot, I'm bashing on the keyboard, and you can see that only a fraction of the screen is updating (this window is pretty small, so the damaged region looks bigger than it might otherwise appear).



Review Of Current (circa M43) Drawing Path

As of M43, when drawing a new frame on Mac, the following sequence of events occurs:

1. The GPU Process draws the entire frame to an offscreen buffer (called an FBO)
2. The FBO is then copied into an onscreen buffer (which is a CAOpenGLLayer)

Of note is that same FBO is used frame after frame.

Of note is that there is no way to specify that only part of a CAOpenGLLayer has taken damage. If anything is to be drawn, then everything will be re-drawn.

First Optimization: Don’t redraw the full FBO

The Chrome Compositor (cc:: stuff) has the ability to track where damage has been taken, but this has never been enabled on Mac. Because the FBO is preserved frame after frame, a first optimization is to have the GPU process only re-draw the region of the frame that changed into the FBO. This still (unfortunately) requires re-drawing the full FBO to the CAOpenGLLayer, but it’s a start.

This is a part of crbug.com/474299, and has been committed already.

Second Optimization: Use Layered CAOpenGLLayers

I mentioned earlier that CAOpenGLLayers must be updated in full -- one cannot specify partial damage to them. This is a strict rule.

That said, it's entirely possible to put CAOpenGLLayers on top of other CAOpenGLLayers. So, one option that we may consider is to, when partial damage is reported, put a new CAOpenGLLayer on top of the old CAOpenGLLayer.

A potential problem here is an explosion of CALayers (because of different damages occurring), but, because we have the full FBO around, we have the option, in the GPU process, of deciding what arrangement of CALayers to keep around (including to just go with a fullscreen layer), whenever we want.

This is getting very Xzibit-like in that it has a whole "yo dawg I herd u like layers and compositors, so I made an compositor output surface that's a tree of CA layers which are drawn by a ui:: compositor that has a cc:: layer tree where some of the ui:: layers are drawn by cc:: compositors which have cc:: layer trees in them ..." going on. But apart from being aesthetically offensive, the idea seems pretty solid.

This is also a part of crbug.com/474299, has functional prototypes, and is nearly production-ready.

Third Optimization: Draw Directly to the CAOpenGLLayer

Another optimization is to skip drawing to the FBO, and instead issue the draw calls directly into the CAOpenGLLayer.

This is a bit against the compositor's architecture. The compositor assumes that it can make a surface current, draw to it, wander out into the weeds for a while, and then later swap the surface. All drawing to the CAOpenGLLayer must be done inside the -[CAOpenGLLayer drawInCGLContext] method, which is a callback made by the system, inside the GPU process.

This means that the entire command buffer segment corresponding to the browser compositor's frame must have its execution delayed until the aforementioned callback is made, and the entire frame draw must happen during this callback. Among other things, this requires that drawing the browser compositor frame have no synchronous calls (glFinish, glGetIntegerv, or even glGetUniformLocation). These requirements are not observed by the browser compositor at the moment, although it may be that they are not hard to add.

This has its own bug, crbug.com/423163, has been semi-functionally prototyped, and is quite a long way from being production-ready.

Of note is that combining this with the Second Optimization is somewhat difficult in that we cannot draw any more than the specified damage rect. Probably the best way to deal with this issue is to ensure that the damage rect be artificially adjusted in the ui::Compositor, to avoid exploding the number of layers.

Preliminary Results Using Power Gadget

The type of page that would benefit the most from the above optimizations is one in which there is a very small part of the window that is animating very rapidly. Canonical examples for this are spinners or blinking cursors (Google Docs, especially).

I constructed a synthetic example with a single, 64x64 rotating layer, to test the effect of these optimizations. This test page animates at 60fps, but the damaged region is only 64x64.

Only the first two optimizations were tested. The third is not ready for testing yet.

In this test, I used the Intel Power Gadget to measure the processor power (which includes CPU and GPU) and the IA power (which just includes the CPU). These measurements were taken every 100 msec for 10 seconds. I then processed the data in Matlab to get a plot of the probability distribution function for the power usage under the baseline, two optimizations, and Safari.

The command line to get these columns from Power Gadget are as follows. Note that this throws out the first 5 seconds, because it takes some time to put focus back to the window with the spinner animation.

```
/Applications/Intel\ Power\ Gadget/PowerLog \
    -file temp.csv -resolution 100 -duration 10 && \
    cat temp.csv | cut -d, -f5,8 | head -n 100 | tail -n 50 \
    > processed.csv
```

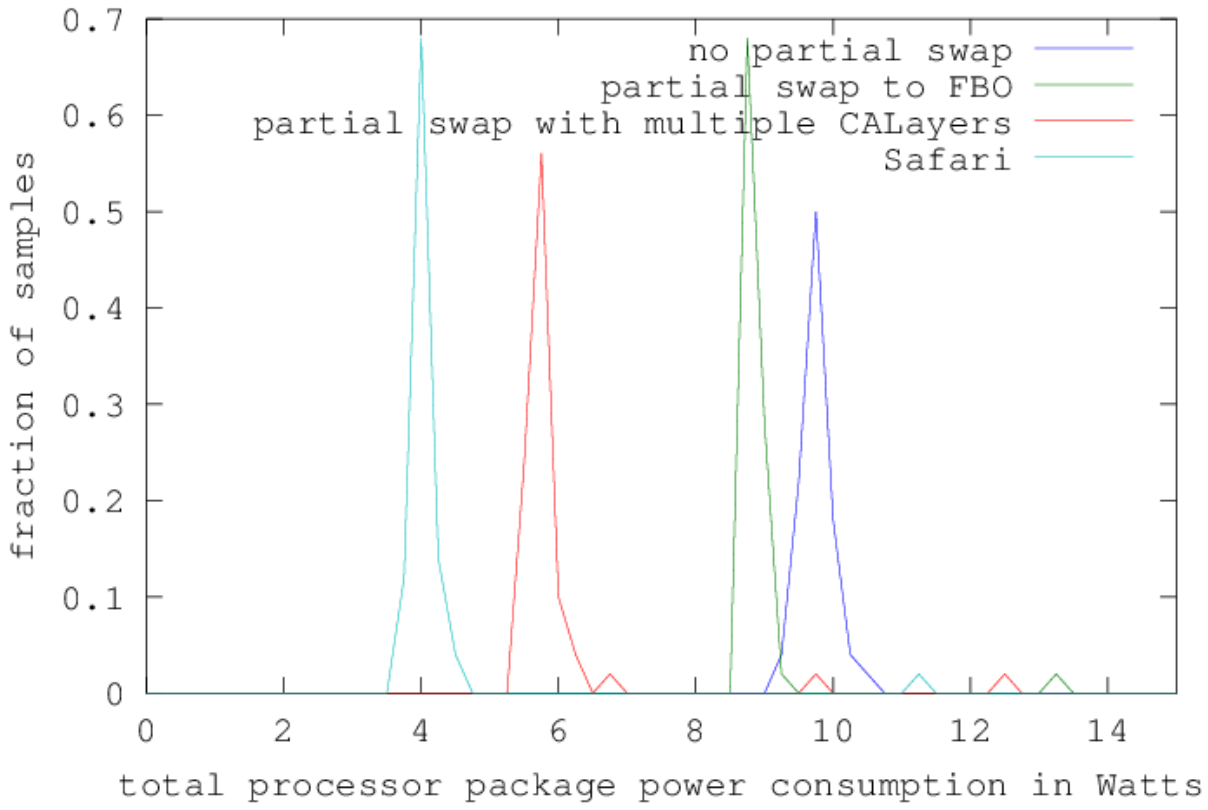
The Matlab code (well, Octave, but whatever) is as follows

```
x = load('watts_spinner_1152_760_no_partial.csv');
y = load('watts_spinner_1152_760_partial_fbo.csv');
z = load('watts_spinner_1152_760_partial_fbo_cropped.csv');
w = load('watts_spinner_1152_760_safari.csv');
n = 50;
xrange = 0:.25:15;

for i=1:3
  figure(i)
```

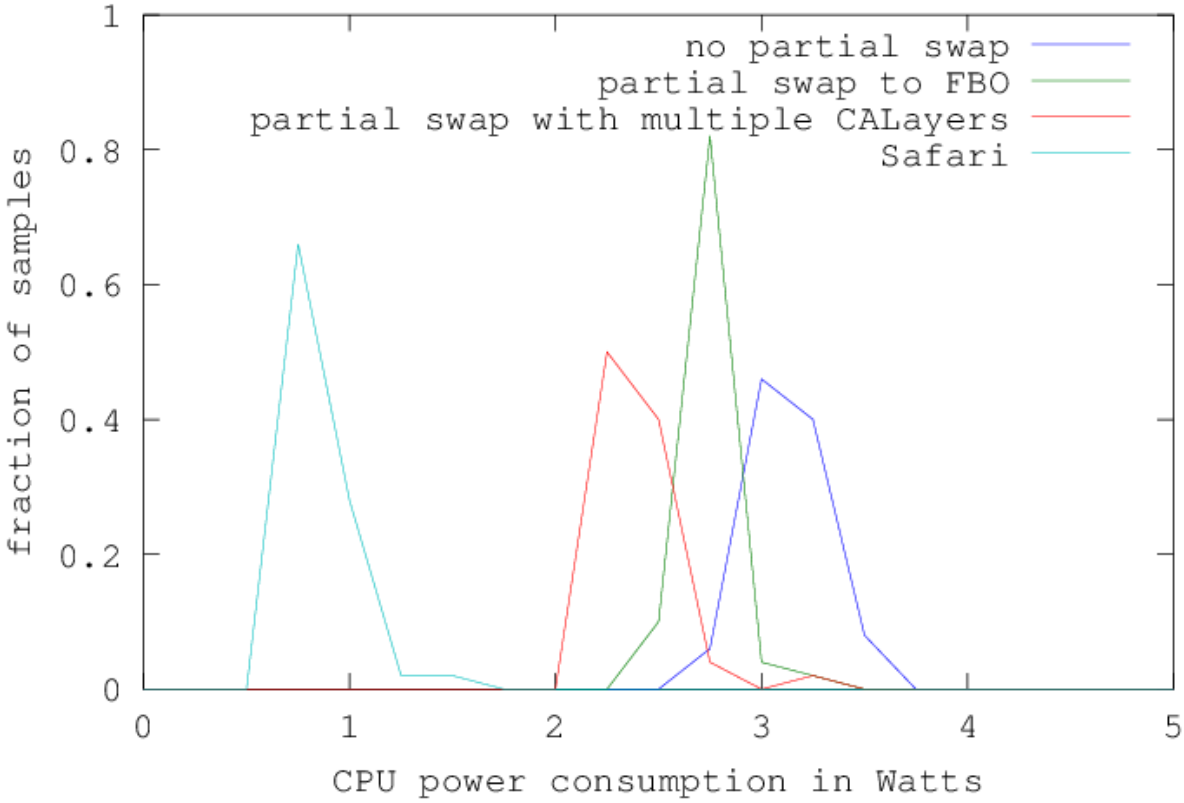
```
  if (i == 1)
    plot(xrange, histc([x(:,1) y(:,1) z(:,1) w(:,1)], xrange)/n);
    xlim([0 15]);
    xlabel('total processor package power consumption in Watts');
    filename = 'power-total.png';
  elseif (i == 2)
    plot(xrange, histc([x(:,2) y(:,2) z(:,2) w(:,2)], xrange)/n);
    xlim([0 5]);
    xlabel('CPU power consumption in Watts');
    filename = 'power-cpu.png';
  elseif (i == 3)
    plot(xrange, histc(
        [x(:,1)-x(:,2) y(:,1)-y(:,2) z(:,1)-z(:,2) w(:,1)-w(:,2)],
        xrange)/n);
    xlim([0 10])
    xlabel('processor non-CPU power consumption in Watts')
    filename = 'power-noncpu.png';
  end
  legend('no partial swap', 'partial swap to FBO',
         'partial swap with multiple CALayers', 'Safari')
  ylabel('fraction of samples')
  print(filename, '-dpng', '-S720,480')
end
```

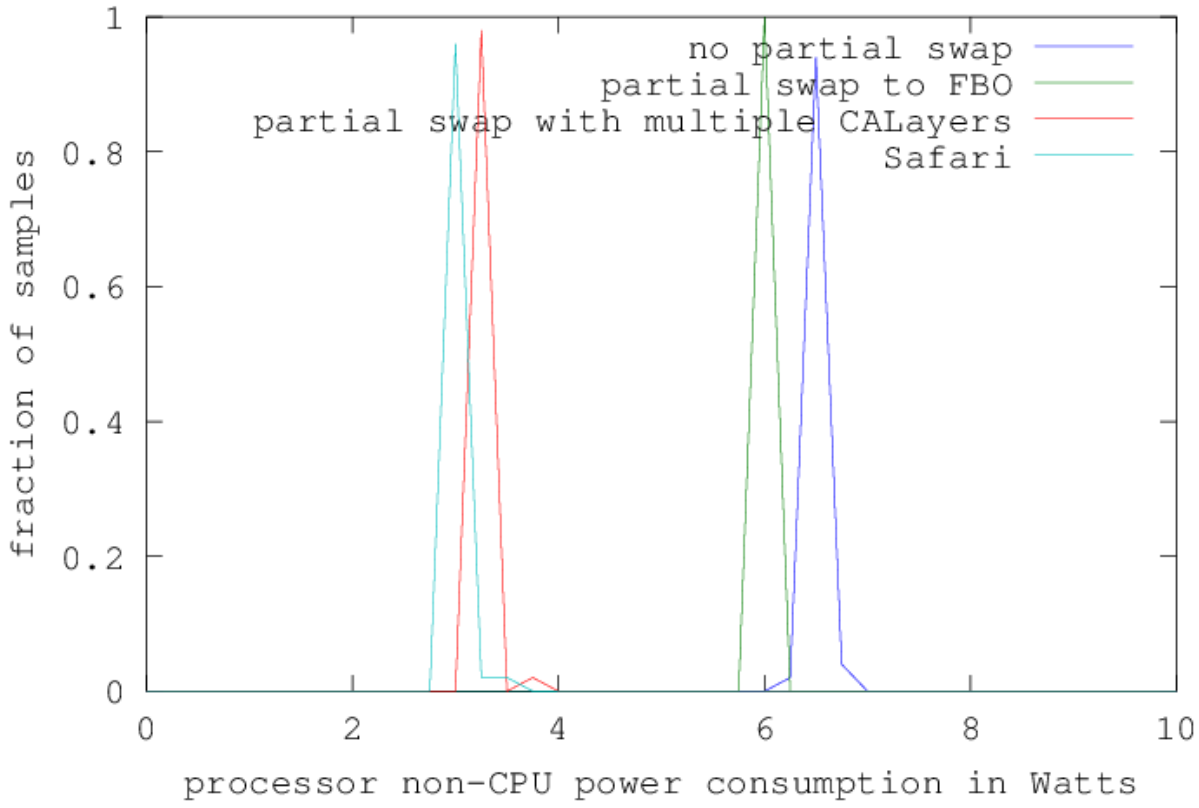
The resulting graphs from running on a retina Macbook Pro (with only the integrated GPU in use) are below:



From this it can be seen that partial swap brings us much closer to Safari in terms of power consumption.

The next graphs break down the above graph into CPU power utilization and non-CPU power utilization.


no partial swap
partial swap to FBO
partial swap with multiple CALayers
Safari
fraction of samples
CPU power consumption in Watts
0
0.2
0.4
0.6
0.8
1
0
1
2
3
4
5



From this it can be seen that seen that the remaining delta between Safari and Chrome (for this page) lies mostly in CPU power consumption.

Feature Prioritization Based on Preliminary Results

Note that none of this covers the third optimization (getting rid of the FBO copy done by the GPU). Also noting that the power difference between Safari and Chrome is almost entirely on the CPU, this suggests that supporting partial swaps is more pressing (for this use case) than removing the FBO copy.

This is a bit expected, as the page being tested was targeted at showing partial swap benefits.