PUBLIC

V8 GC Parallelization Issues

Author: gab@
Last update: 2018-01-23

This document investigates issues in Chrome 66.0.3328.2 with parallel GC where we could do better in V8 (with or without the collaboration of the TaskScheduler).

For a broader description of the parallel work being submitted to TaskScheduler by V8, refer to this document.

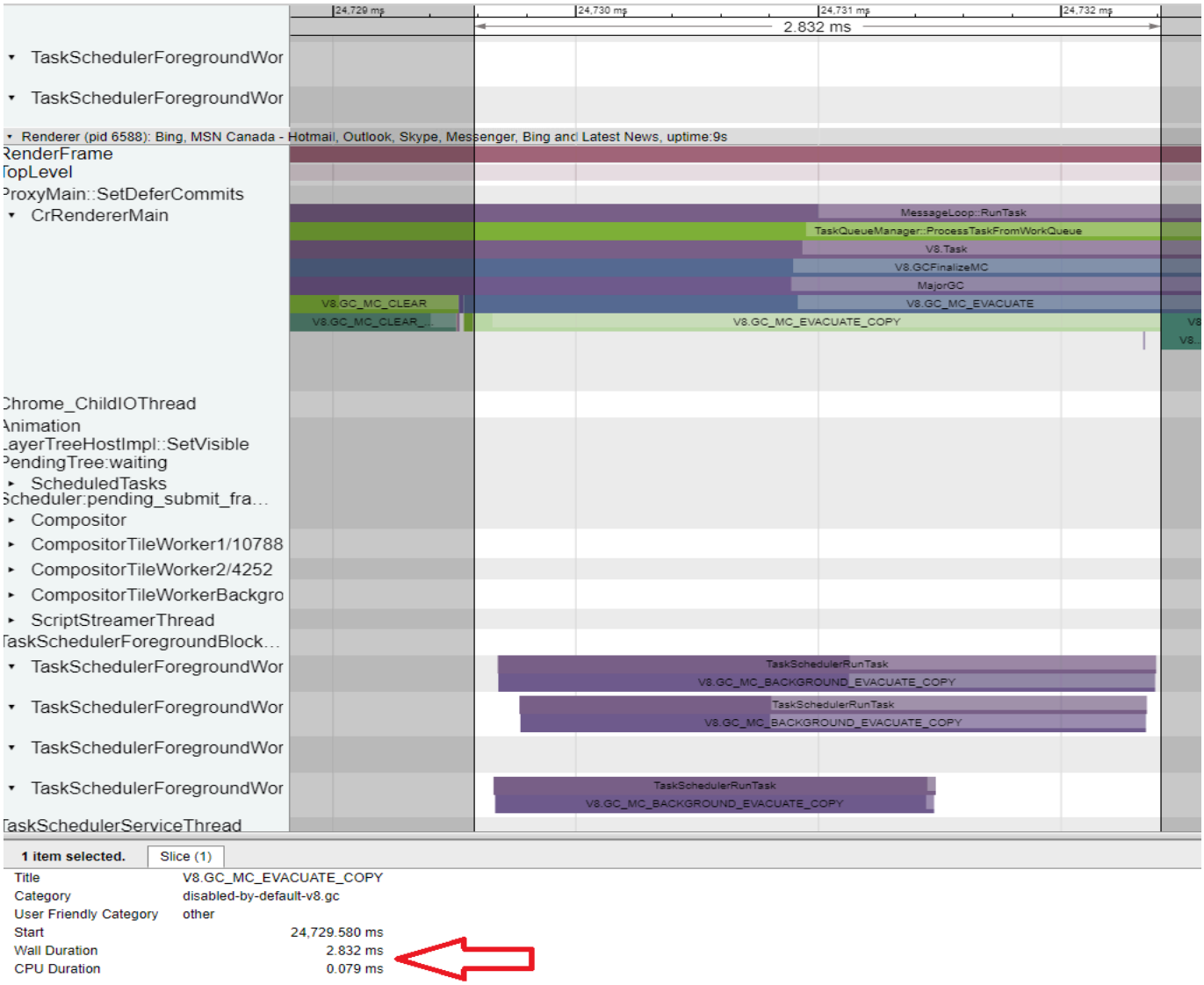
Full traces can be downloaded here on demand.

Tip: each bar in the trace view is colored with a darker part whose size is proportional to the time spent running on the physical CPU (as opposed to wall-time).

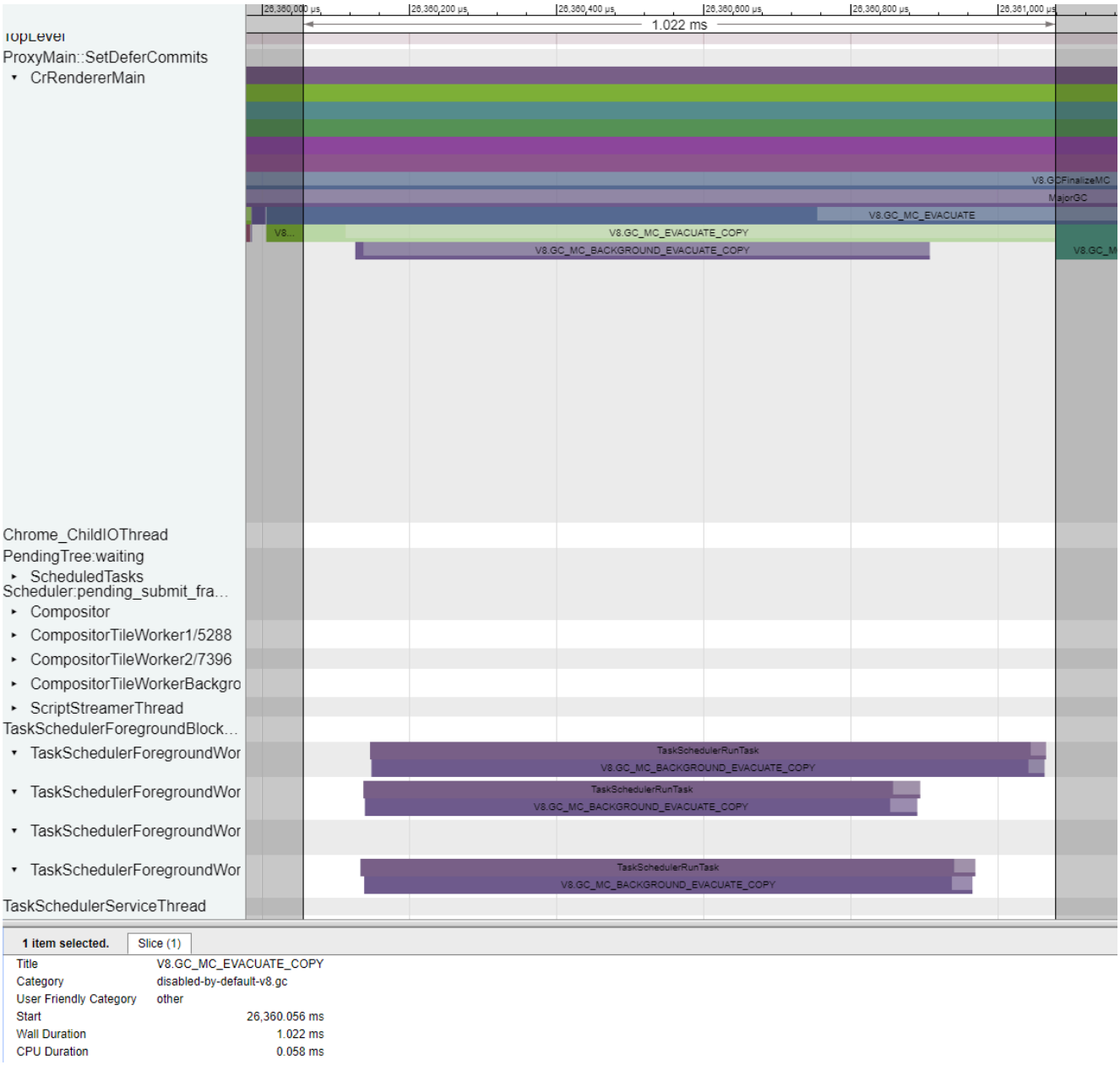
Issues #1

Main thread not contributing to evacuate

GC_MC_EVACUATE_COPY on the main thread kicks off parallel tasks but doesn't contribute itself:

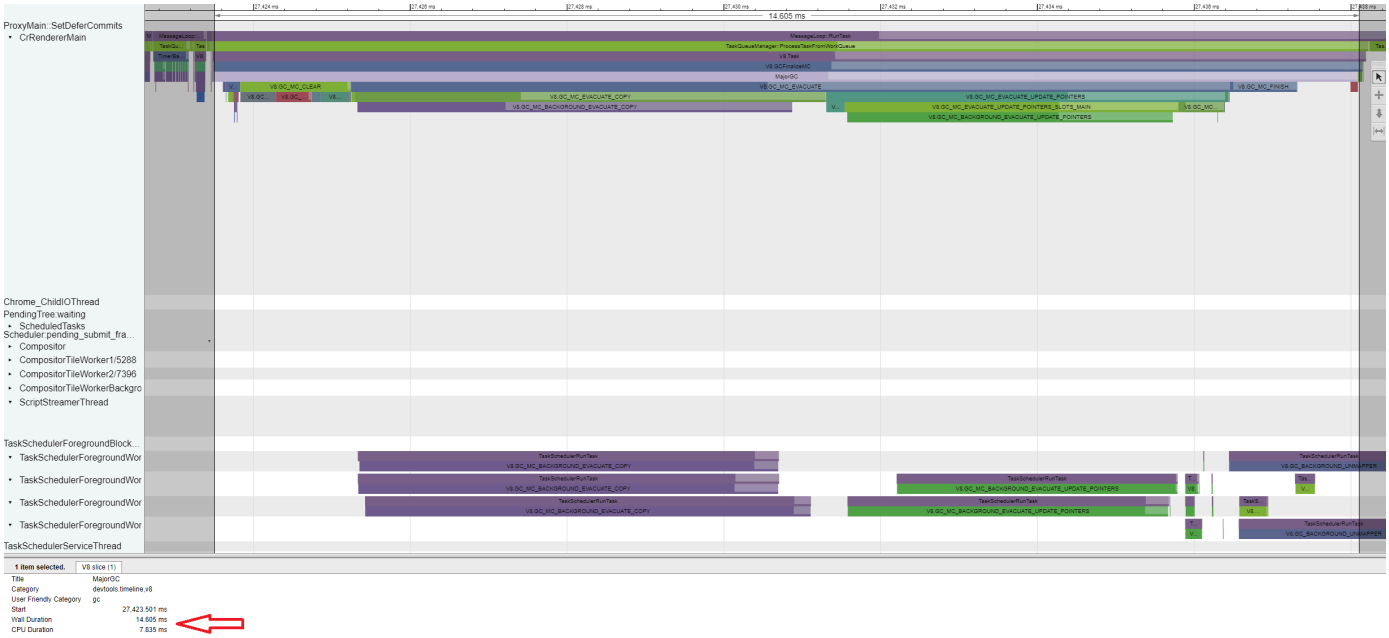


Here's another one where it managed to contribute but was still idle for 95% of it (0.05ms CPU time vs 1ms wall time).



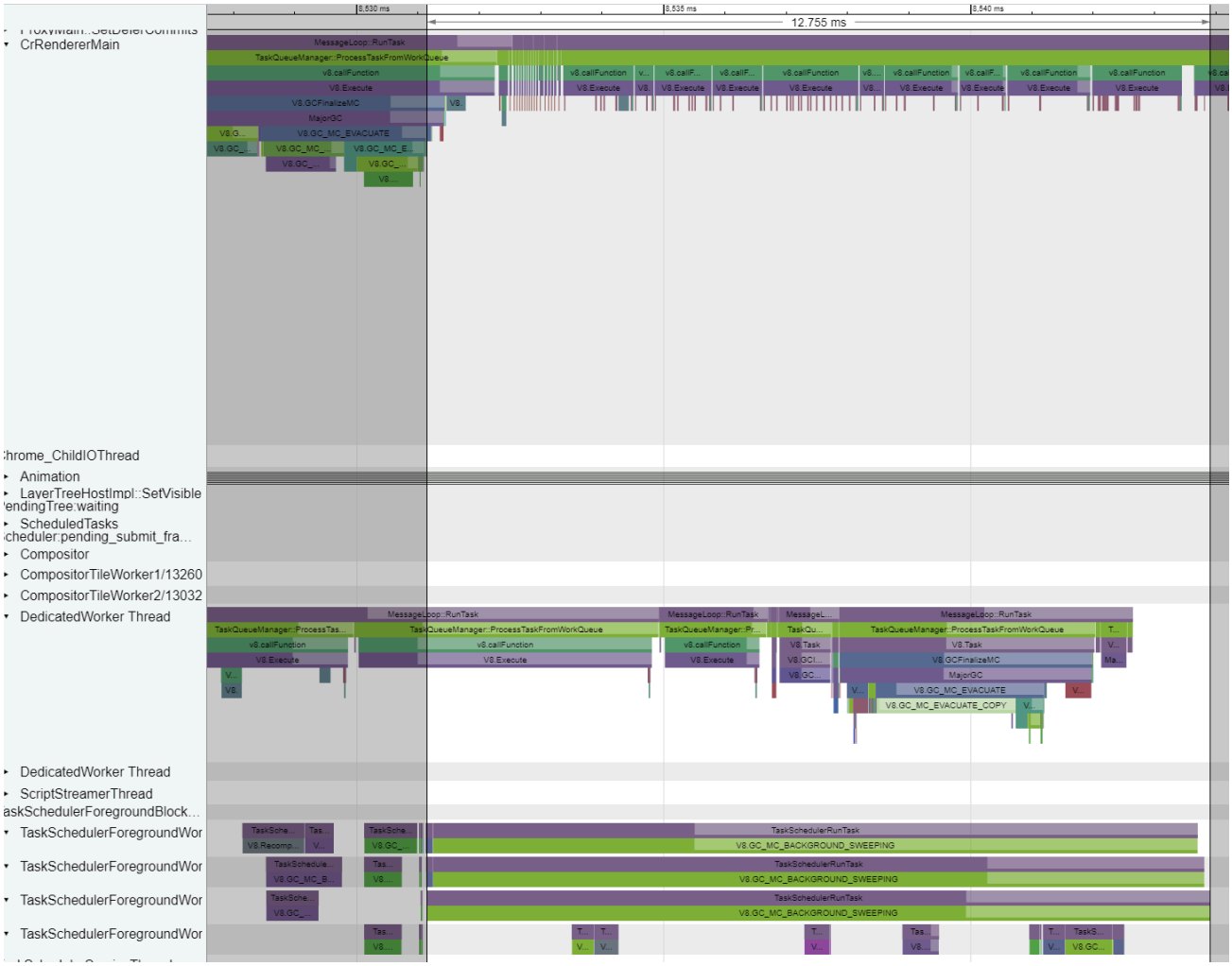
Main thread idle for 50% of a 14ms GC

This would be okay if all threads were descheduled (e.g. because this is a background tab running at background process priority). But this isn't the case because workers are getting a full CPU load (and the rest of the trace shows nothing is happening in other chrome processes).



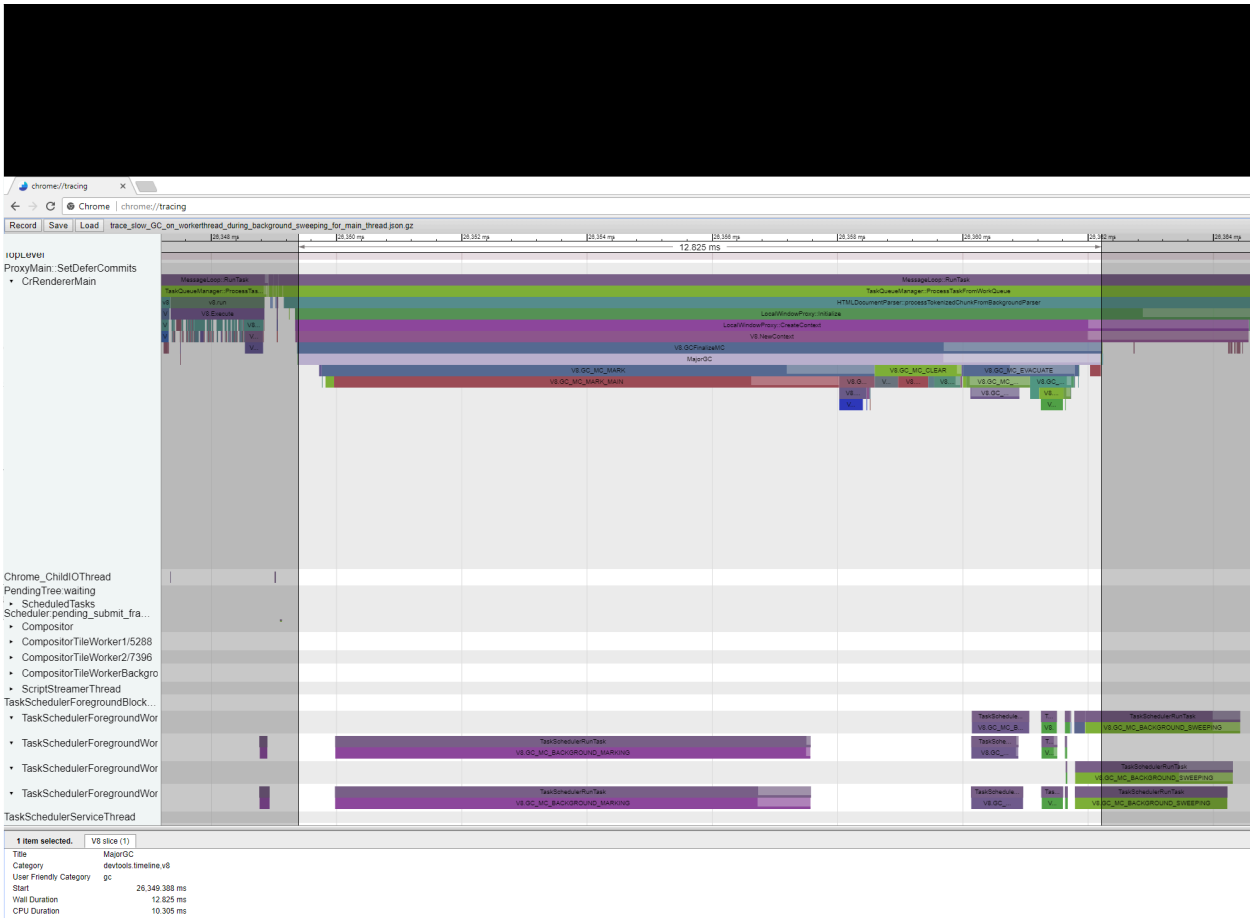
Slow GC in one isolate because other isolate stole all workers

In this case it’s a worker thread’s GC being slow because main thread is doing concurrent sweeping, but it could be the opposite (main thread slowed down by worker thread’s isolate; and any concurrent operation can technically be problematic).

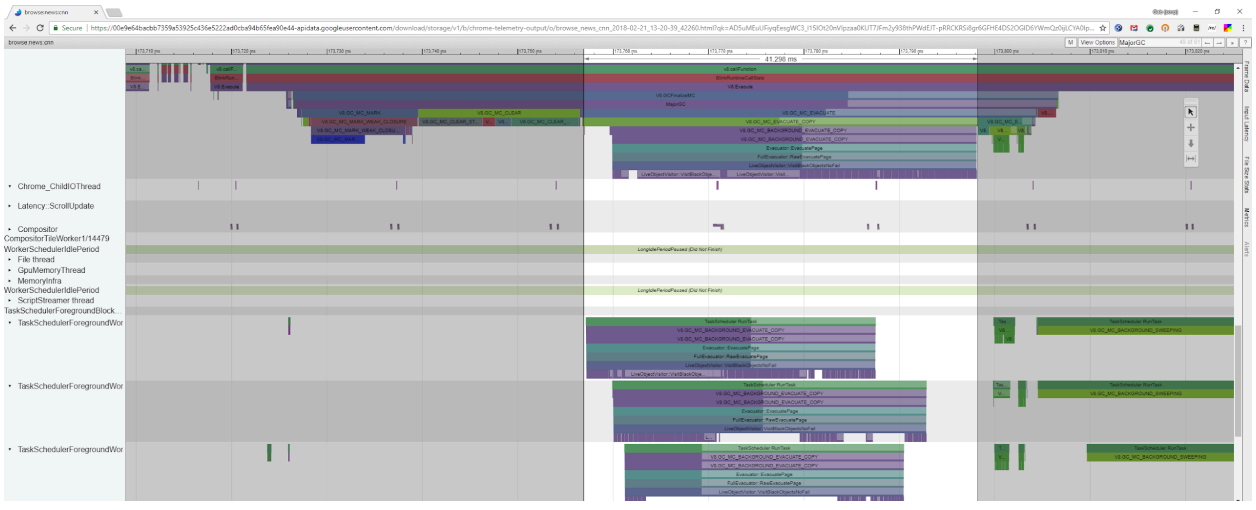


Not enough workers for parallel marking

This is a 4-core machine and one core is unutilized for an 8ms marking phase in a 12ms MajorGC (note to self: we should probably make “num workers == num cores - 1” since main thread should always be busy).



Other operations (i.e. all but evacuate copy) aren't desched so it doesn't appear to be kernel/test machine's fault



Solution #1

Smoother distribution of assignments in task array

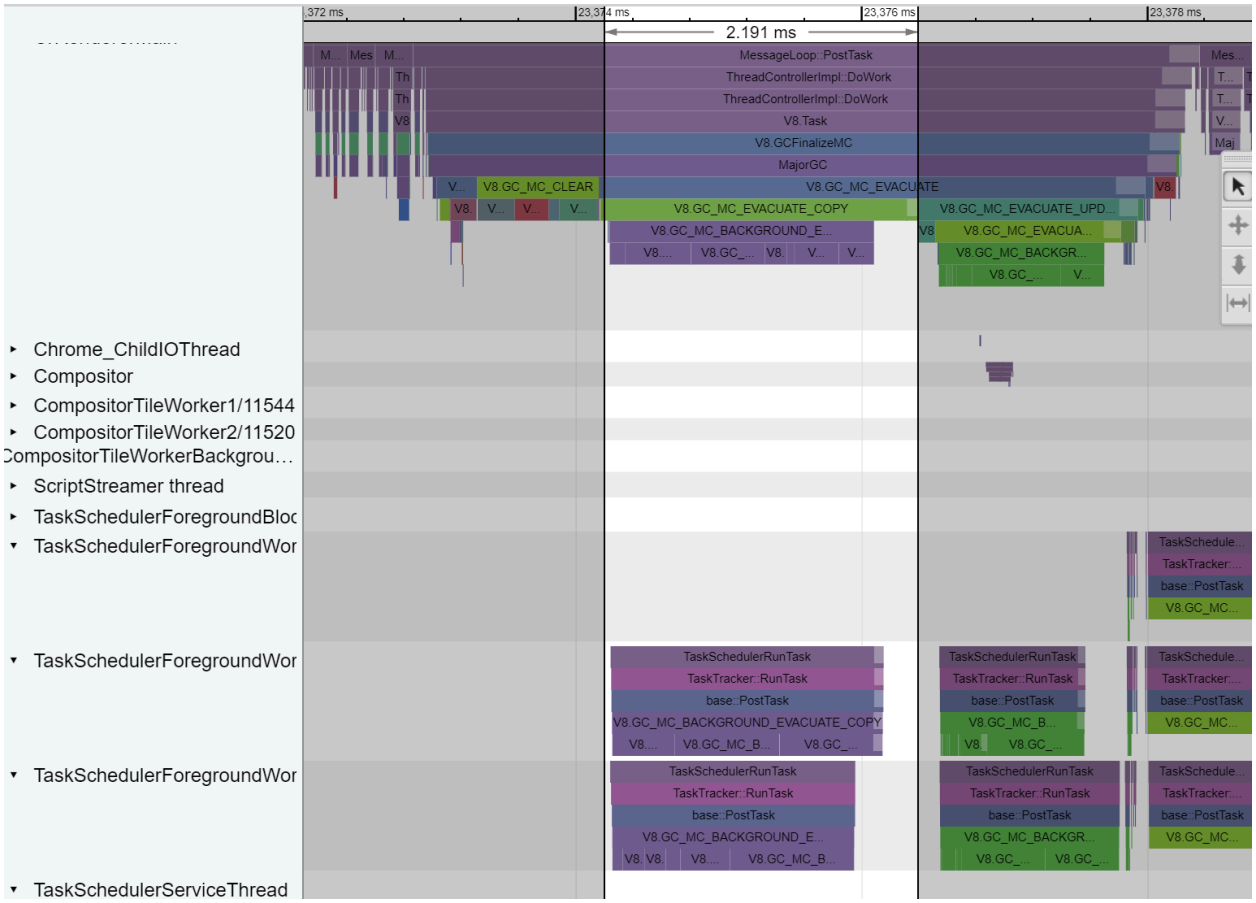
Discovered that the algorithm to assign job to each worker resulted in a lot of overlap (and hence in the main thread often merely getting to contend with other workers assigned the same workload and already ahead of it).
In see below we see change's effect on dev machine: main thread contributes full portion of its assignment (but finishes much earlier in this case so still waits but definitely better overall -- and less contention).


3.234 ms
3.236 ms
3.238 ms
3.240 ms
5.485 ms
CrRendererMain
MessageLoop::RunTask
TaskQueueManager::ProcessTaskFromWorkQueue
V8.Task
V8.GCFinalizeMCReduceMemory
MajorGC
V8.GC_MC_CLEAR
V8.GC_MC_EVACUATE
V8.GC_MC_EVACUATE_COPY
V8.GC_MC_B...
Chrome_ChildIOThread
Compositor
CompositorTileWorker1/129600
CompositorTileWorker2/118648
CompositorTileWorker3/131364
CompositorTileWorker4/119044
DedicatedWorker Thread
TaskSchedulerForegroundBloc
TaskSchedulerForegroundWor
TaskSchedulerRunTask
TaskTracker::RunTask
V8.GC_MC_BACKGR...
TaskSchedule...
TaskTracker:...
V8.GC_MC_B...
V8.GC_MC_BACKGROUND_E...
V8.GC_MC_BACKGROUND_EVACUATE_COPY
Renderer (pid 127748): Google Maps, uptime:38s
CrRendererMain

Issues #2

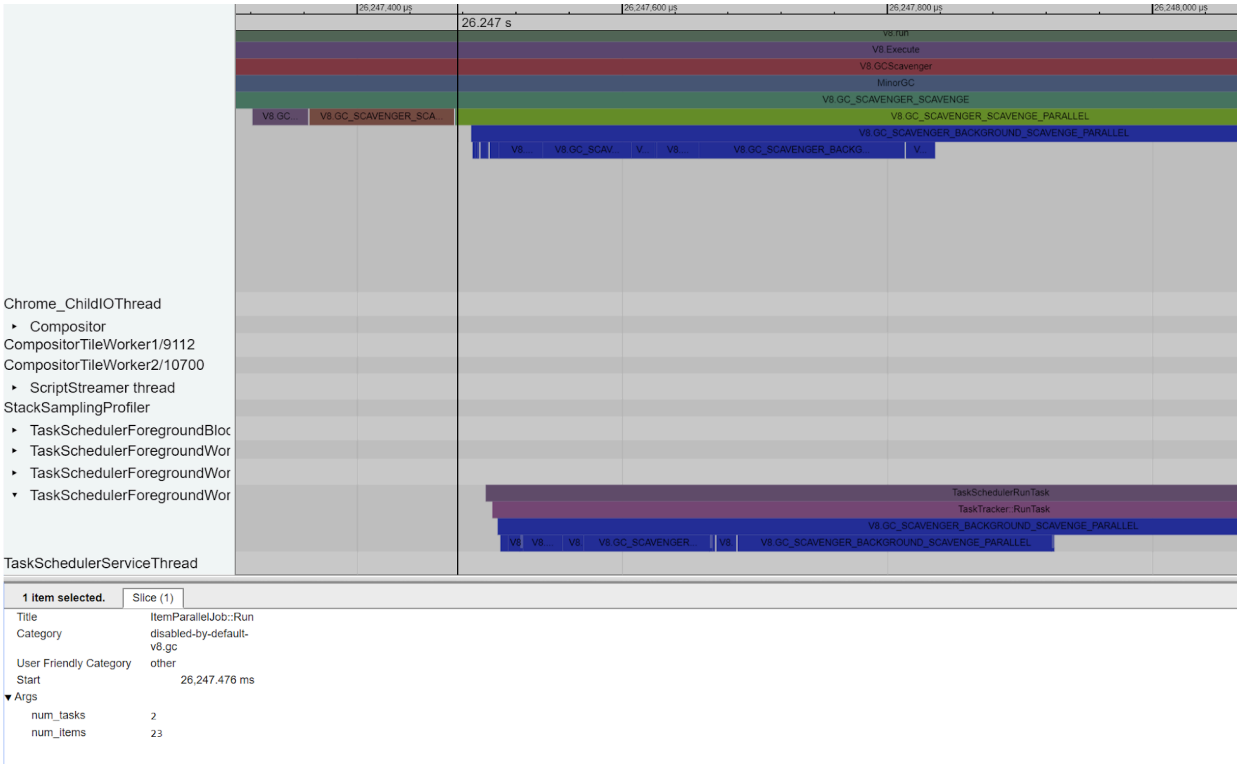
Many work items, not enough tasks

Aaah, misconfiguration issue… (fixed: CL1, CL2). Had tweaked TaskScheduler to have one less worker than cores (assuming busy main thread as noted above). But GC was using “NumAvailableBackgroundThreads()” as a signal for “num cores”. Manually doing these workload computations on opposite ends of the codebase is error-prone… hinting once again that it’d be much simpler to have a job’s API exposed by TaskScheduler directly.



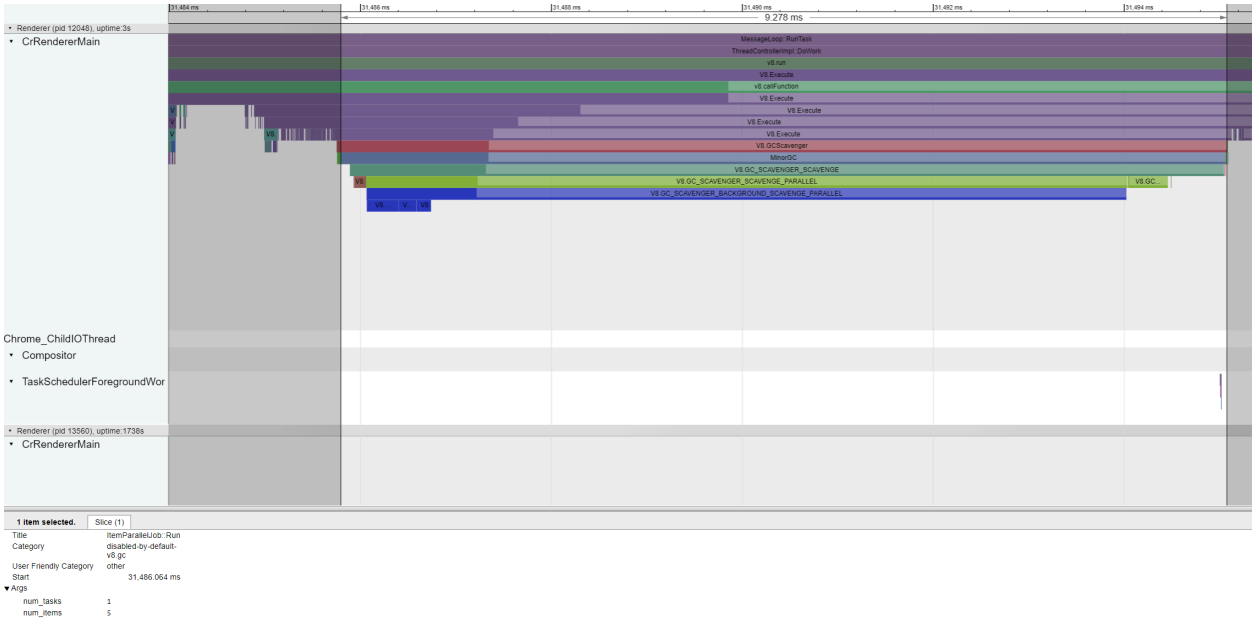
Scavenger step with 23 work items gets only 2 tasks

Note: Have anecdotally seen Scavenge steps with ~160 work items use only 1 task (because task count is based on page size -- may want to revisit that eventually).



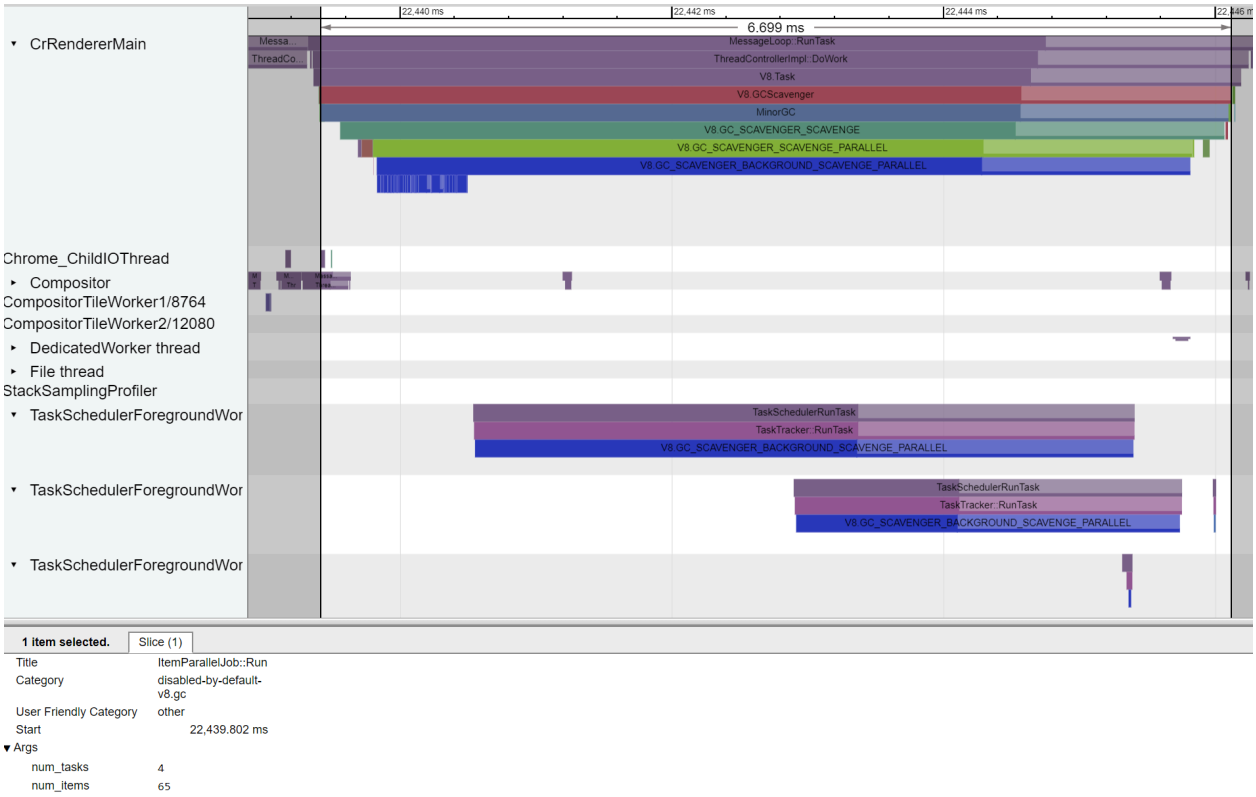
Poor sharding of Scavenger work (5 large items, 1 task)

Lots of desched, inefficiently fetching pages from RAM one by one in a single task?



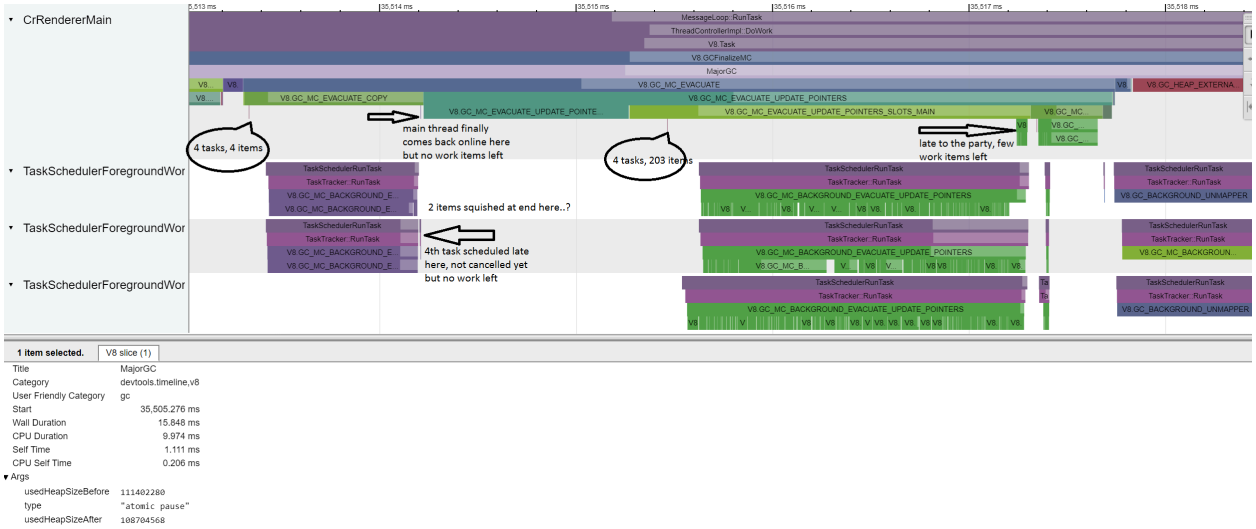
Poor latency in otherwise properly sharded Scavenge (4 tasks)

Suspicious… added more task_scheduler tracing events to diagnose this if it happens again…



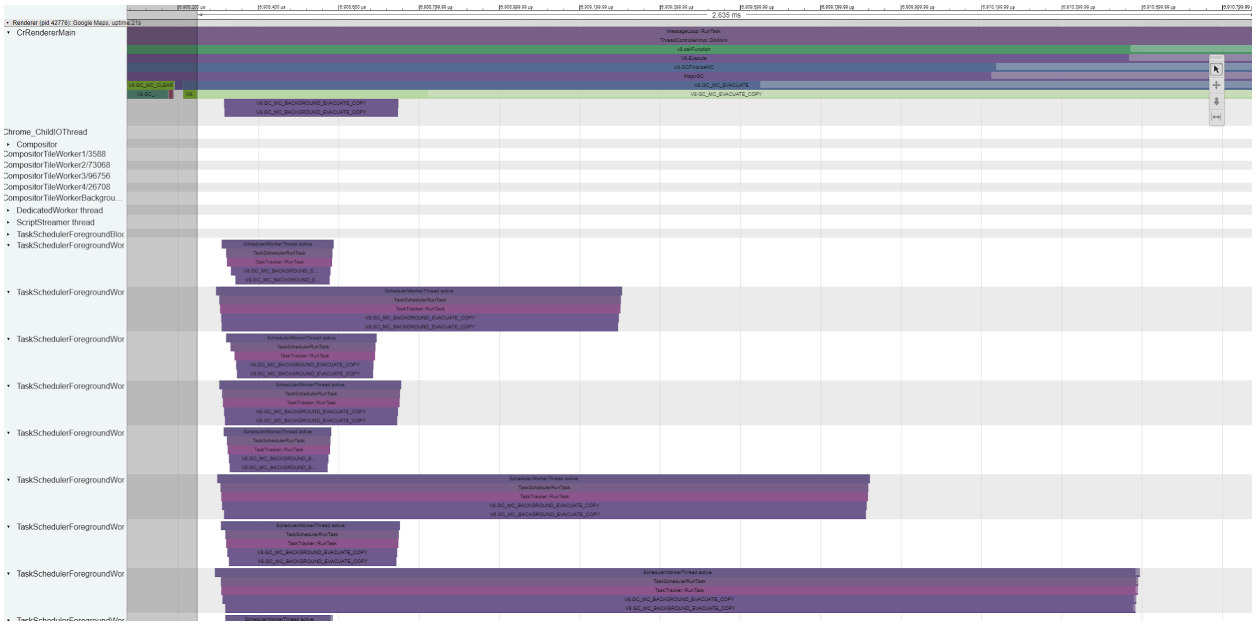
Main thread suspiciously idle for 6ms during 15ms MajorGC (CPU not going above 50% and work items available)

Note: Recently added trace entries for individual work items, makes us clearly see that main thread went idle instead of picking up available work items here..?
And in the evacuate copy phase, for some reason, TaskScheduler didn’t schedule the 4th task on the extra worker?
This might have been a background tab at background OS priority, is the kernel somehow restricting the number of active threads in a background process? Note to self: would be nice to add more details to tracing around backgrounding.



Evacuate_Copy work items too coarse

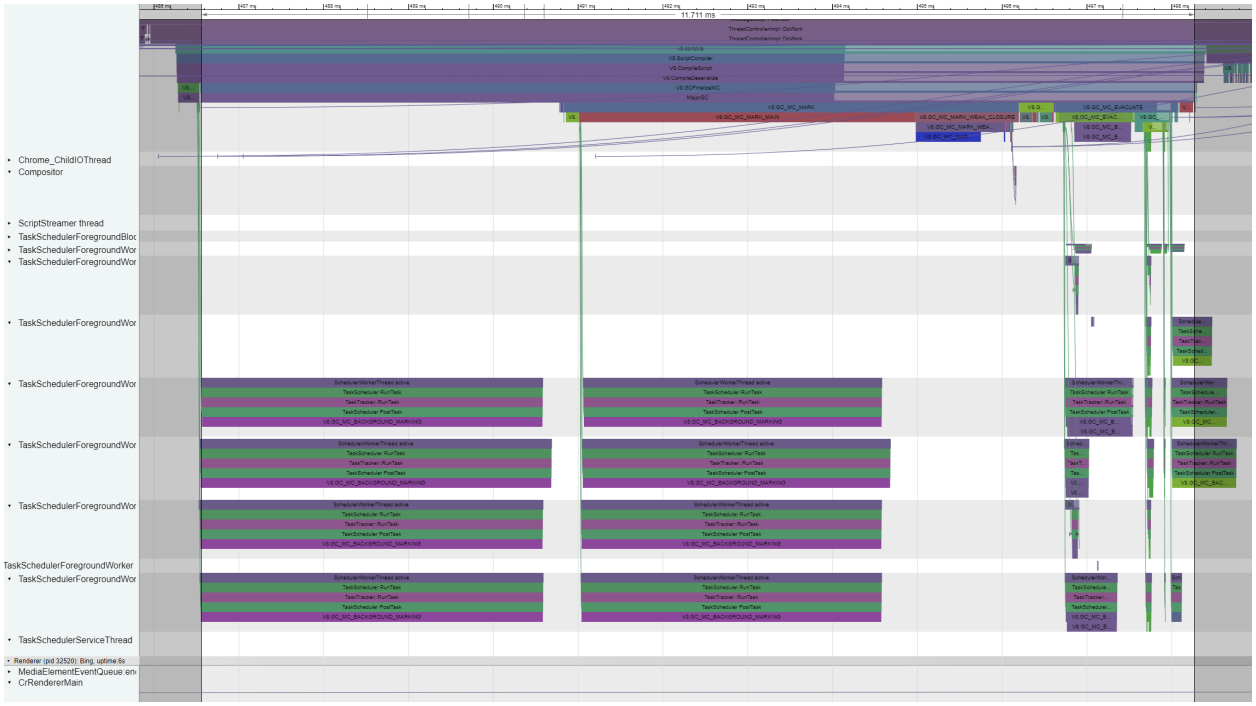
Poor distribution of workload (2.5ms evacuate_copy, 0.5ms contributed by main thread and most other workers).



Main thread blocks on concurrent marking workers when MajorGC kicks in without joining them

If incremental (concurrent) marking happens to be under way. MajorGC begins with the main thread blocking on its completion, without helping workers. This can be particularly problematic

on ARM where we know the main thread tends to be on a BIG core while the workers are usually on little workers.

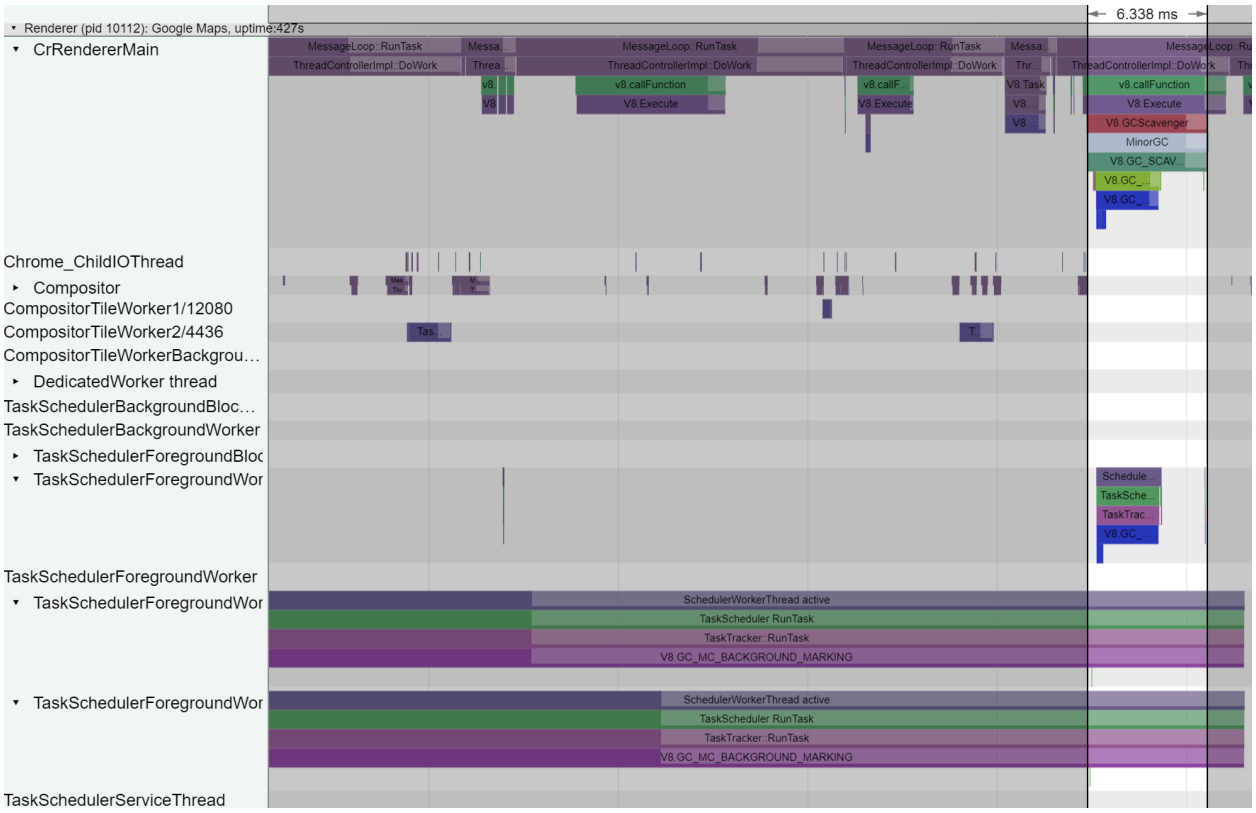


Concurrent Marking could be more aggressive

It currently only uses num_cores/2. A change of mine accidently brought this down to (num_cores-1)/2 and we saw [regressions on many perf graphs] (probably down from 2 to 1 cores as machines typically have 4 cores. This was recovered but I now assume that using (num_cores-1 == 3) cores for concurrent marking would also result in further improvements. Need to have smaller/preemptable work items first I assume however?

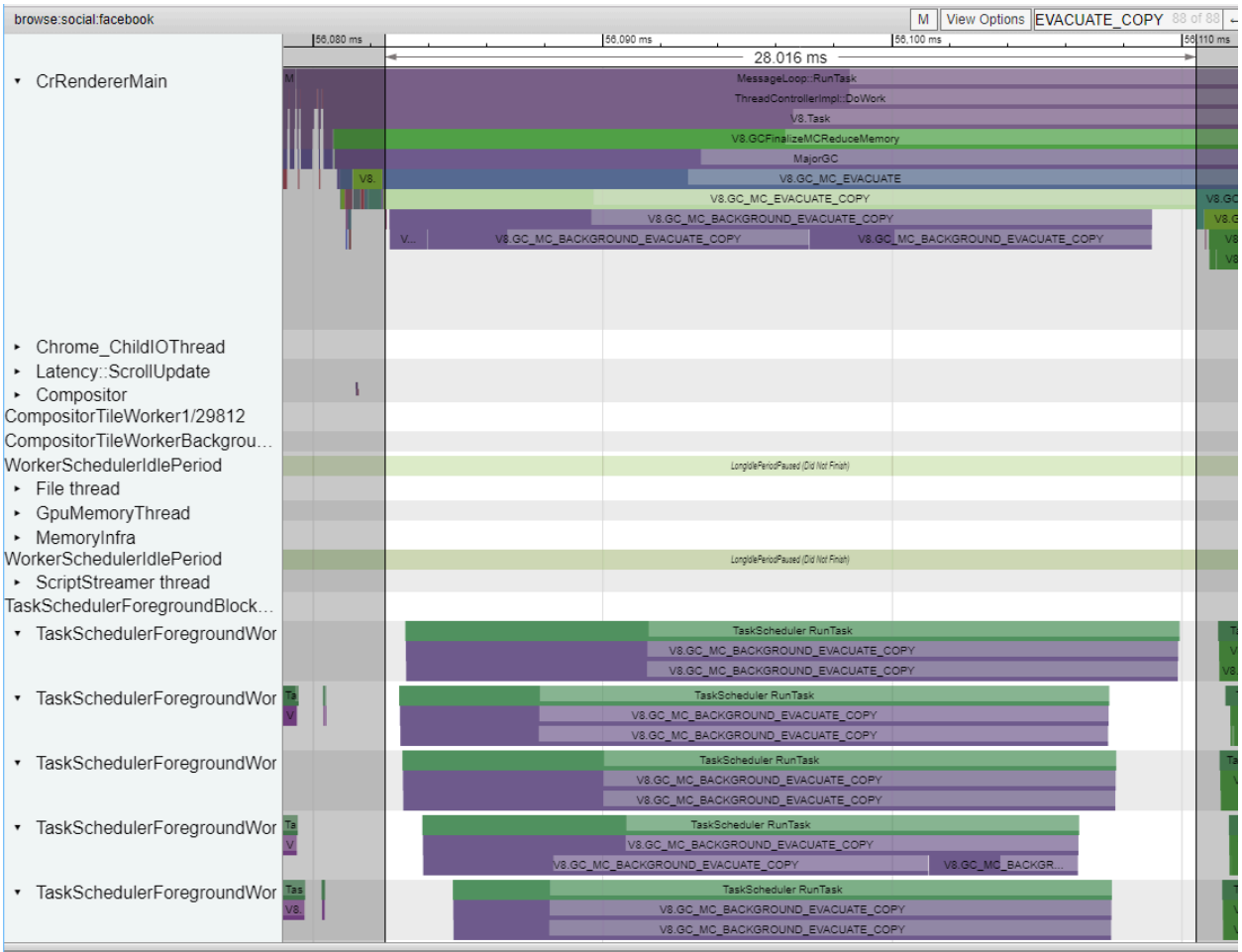
Pausing concurrent marking steals the workers from the pool

Fixing this will also allow us to be more aggressive with concurrent marking.
While this screenshot only shows a 6ms MinorGC with two inactive workers, it shows what's happening and one could imagine much worse scenarios of this happening in the wild.



Suspiciously inactive EVACUATE_COPY

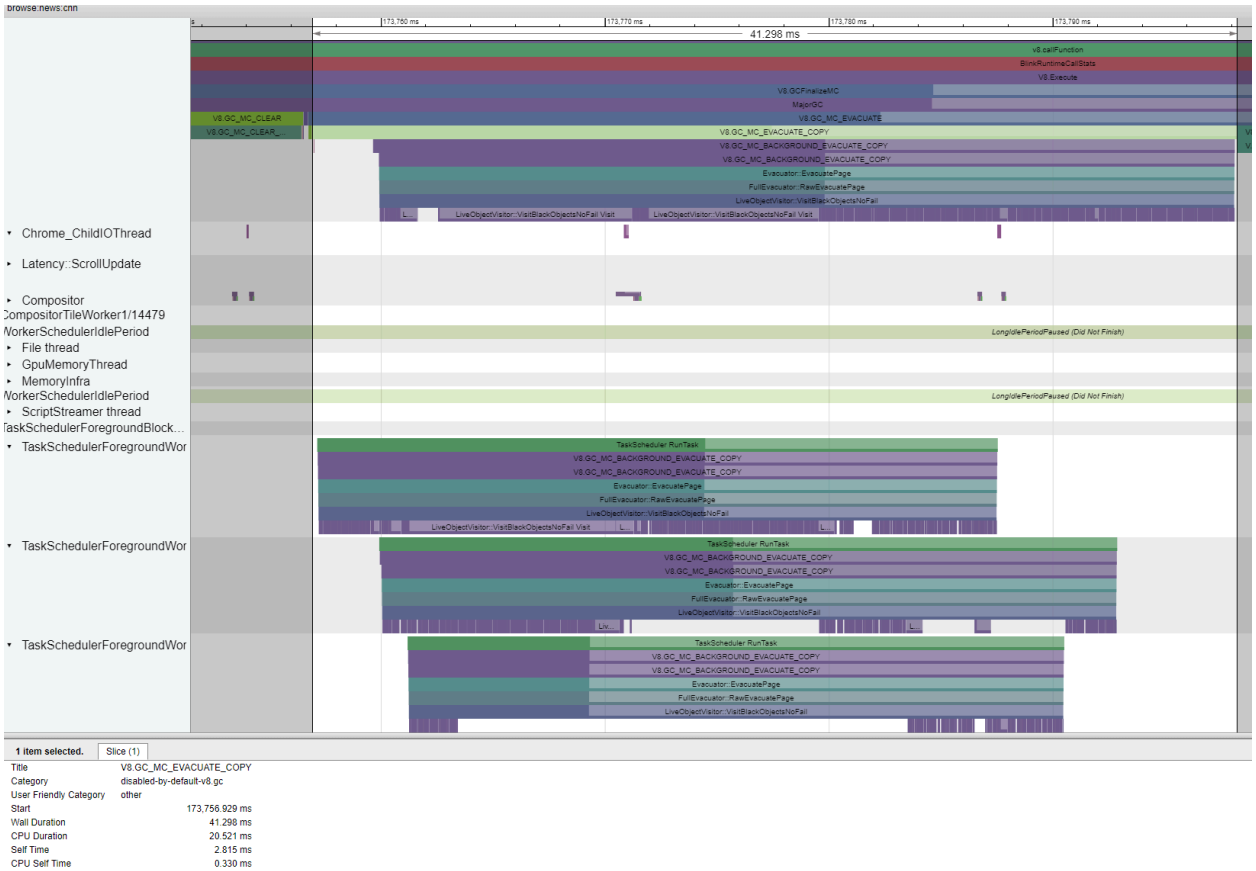
CPU idle for 21/28ms on main thread and worker threads.



1 item selected. Slice (1)

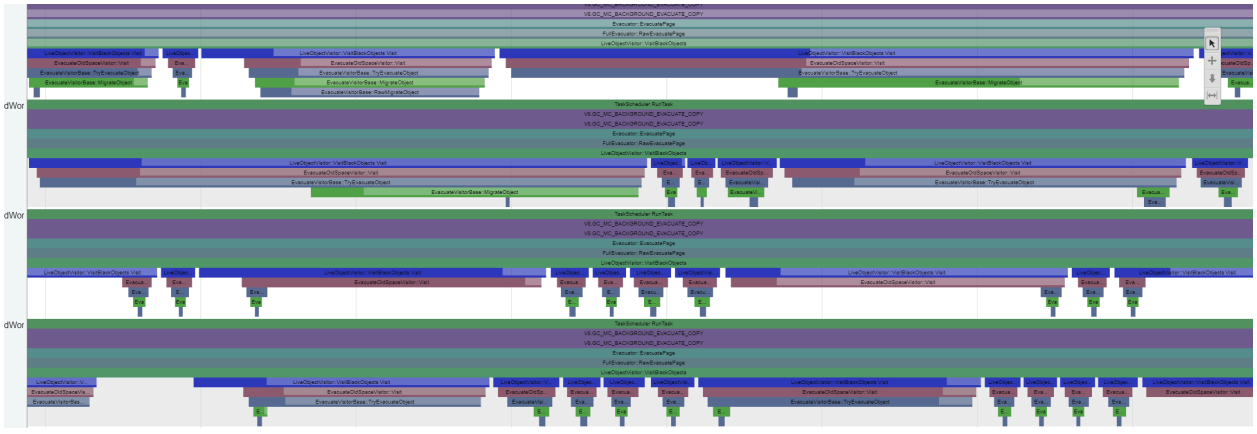
Title	V8.GC_MC_EVACUATE_COPY
Category	disabled-by-default-v8.gc
User Friendly Category	other
Start	56,082.478 ms
Wall Duration	28.016 ms
CPU Duration	7.222 ms
Self Time	1.678 ms
CPU Self Time	0.247 ms

Finer grain



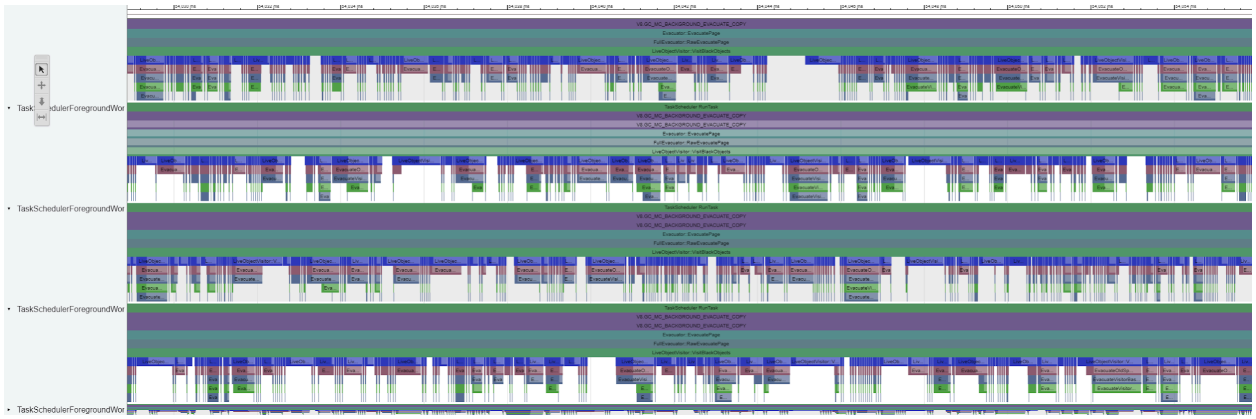
Even finer grain evacuate copy steps (some interrupted, some 100% CPU)

Interesting that in some slow steps : EvacuateVisitorBase::RawMigrateObject completes early.



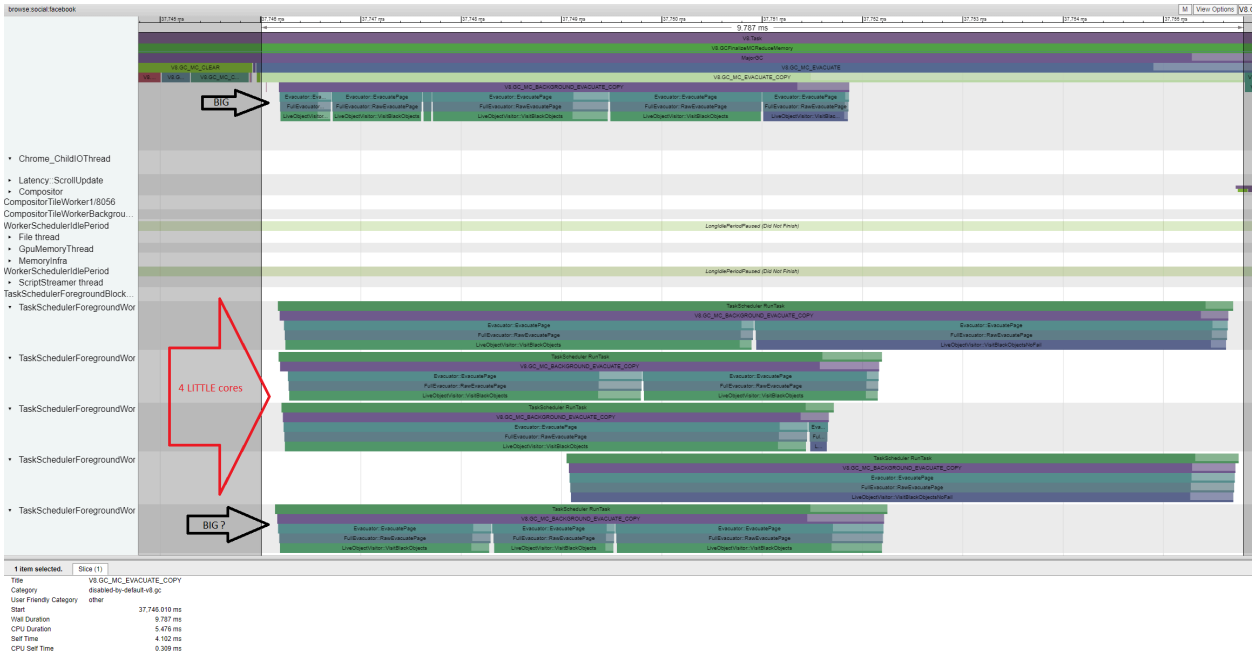
[Analysis]

Broader view



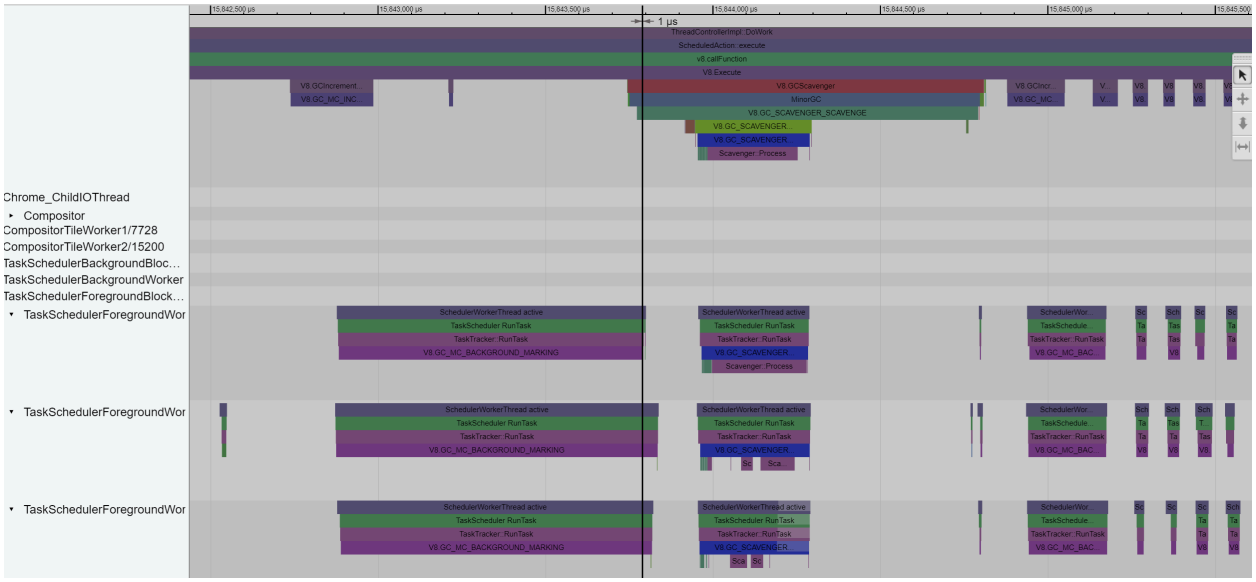
BIG.little GC priority inversion

The main thread is on a BIG core along with one worker (seems like). 4 workers our on little cores and the main thread ends up waiting on them.

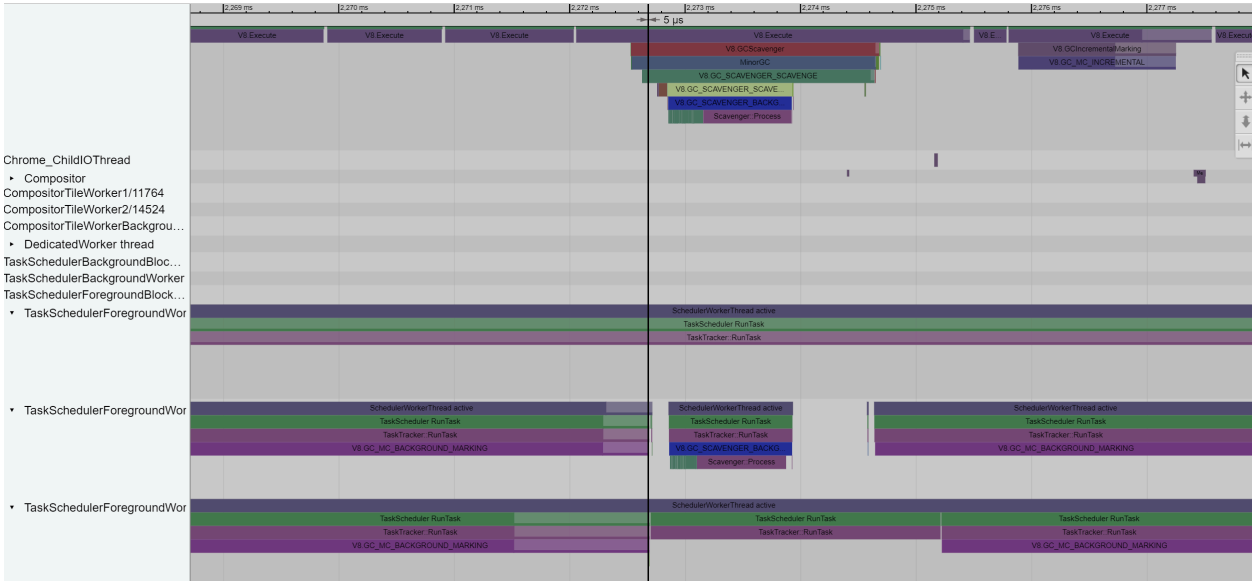


Solution #2

Preempt concurrent markers when Scavenge kicks in. Previously we would pause concurrent marking but wouldn't yield the worker thread back to the pool as such Scavenge would typically not have workers to help it.



Here’s another interesting trace which shows non-GC related tasks contending for the workers as well. Ideally these would be pieced in finer grained tasks to be able to yield to high-priority main thread blocking scavenge tasks.



Related Tracing Improvements

Flow events for TaskScheduler tasks