Performance Analysis of Lazy compilation for WebAssembly

author: ahaas@
date: 2022-01-25

Background

V8 eagerly compiles all functions of a WebAssembly module. Thereby V8 does not only compile the function which will get executed but also many functions that will never get executed. For Photoshop, Liftoff generates 184MB of code on x64. For Google Earth it is 50MB code, and for a Unity game it's 57MB. The advantages of eager compilation are that

- all the compiled code is available immediately. There is no threat of having jank at some point because functions need to be compiled;
- background threads can help with the compilation;
- code space can be written in big chunks, which reduces the number of times the permissions of code space have to be changed to be writable.

An alternative to eager compilation would be lazy compilation, where functions only get compiled when they get executed for the first time.

Memory reductions of lazy compilation

With lazy compilation the memory consumption of generated code could be reduced significantly. For Photoshop, memory consumption could be **reduced by 79%** to 39MB, for Google Earth the **reduction** would be **50%** down to 25MB, and for the Unity game the **reduction** would be **68%** down to 18MB of code.
The memory reductions are also visible in the telemetry benchmark of Photoshop and AutoCAD (Thanks to bikineev for helping me with the metrics). The following Pinpoint jobs measure the performance impact of enabling lazy compilation:

- Photoshop on Linux perf;
- Photoshop on Linux perf; system_health.common_desktop
- AutoCAD on Linux perf;
- Photoshop on Mac M1 mini;
    - Unfortunately there is no baseline measurement on the Mac.

One of the key memory metrics, the renderer's **Private Memory Footprint**, improved significantly. For Photoshop we see a **reduction of 38%**(452MB -> 279MB). For AutoCAD the **reduction is 36%** (237MB -> 150MB).

Performance impact

One main disadvantage of lazy compilation is that all functions get compiled on the main thread. With eager compilation, all compilation can be done on background threads. Additionally, with streaming compilation, eager compilation can do all the compilation during the download, whereas lazy compilation would compile all functions after the download.
The advantage of lazy compilation is that often a big number of functions never gets executed, or they don't get executed during startup. This can lead to faster startup time with lazy compilation.

Photoshop, which uses streaming compilation and also compiles significant amounts of code, shows a slower startup time with lazy compilation. On my Windows laptop, with an Intel(R) Core(™) i7-7600U CPU @ 2.80GHz, 2904 Mhz, 2 Cores, 4 Logical Processors, I measured startup times of 22.742 sec and 22.344 sec with eager compilation, and 24.564 sec and 22.983 sec with lazy compilation. I did not do a proper measurement, and did not provide a clean test environment. As a tendency it seems that lazy compilation introduces some startup time regressions for Photoshop, but these regressions don't seem to be significant.

On Google Earth, there is no visible performance impact of using lazy compilation. For the Unity game there are some improvements and some regressions, see the table below.

Detailed result in the Unity game, measured on my Windows laptop. Additionally there is a Pinpoint run on Linux perf.

	Dynamic Tiering	Eager Compilation	Lazy Compilation
numRenderedFrames	1000	1000	1000
totalTime	47344.05	49301.7	44704.5
totalRenderTime	38880.55	39851.4	36582.8
wrongPixels	0	0	0
webGLCpuTime	0	0	0
webGLApiCallCount	0	0	0
cpuTime	29238.3	30454.3	24468.3
cpuIdle	22.63953301	22.25681759	31.24715953
fps	25.74033099	25.20141904	27.3410043
pageLoadTime	8437.5	9416.95	8067.7
numStutterEvents	171.5	135	254
usedJsMemory	365032565.5	364865270	364923869
excessFrametime	15550.45	17030.55	10817.35

pageLoadTimeToFrame1	8463.5	9450.3	8121.7
pageLoadTimeToFrame10	8624.4	9637.9	8303

Legend:
totalTime: How long the whole test took, from page load to finish. This includes XHR download times, so net speed affects this (msecs).
totalRenderTime: How long time it took from the *end* of the first rendered frame to the end of the last rendered frame (msecs).
wrongPixels: The number of pixels that failed the per-pixel reference image test.
result: Overall result, either PASS, FAIL (finished but reftest failed) or ERROR (run aborted on exception).
webGLCpuTime: The total CPU time spent inside browser's WebGL API functions (msecs).
cpuTime: The total CPU time spent inside page code in requestAnimationFrame()s (msecs).
cpuIdle: The fraction of animation time that the CPU was not executing user requestAnimationFrame() code (%). This assumes that all time outside rAF() is practically "idle" time.
fps: Total frame rate averaged throughout the whole run (frame/second).
pageLoadTime: How long it took from page load to get to the beginning of rendering the first frame (incl. downloads, compilation, parsing and app main()) (msecs).
numStutterEvents: The number of rendered frames that took abnormally long to complete compared to their previous frames.
openIndexedDB: How long it took to open an indexedDB database connection (msecs).
indexedDB.get(): How long all page IndexedDB read operations took accumulated (msecs).
indexedDB.put(): How long all page IndexedDB store operations took accumulated (msecs).
XMLHttpRequests: How long all page XHR download operations took accumulated (msecs).
WebAssembly.compile(): How long WebAssembly Module compilation took (msecs).
WebAssembly.instantiate(): How long WebAssembly Module instantiation took (msecs).
usedJsMemory: Number of JavaScript heap bytes in use. (0 if browser does not report this).
shaderCompilation: How long it took to compile all page shaders (msecs).
readDataBlob: The time it took to read the main asset .data file blob from disk to memory.
main(): The time taken in executing the initial application main() function (msecs).
excessFrametime: How much of overall time was spent above the "sweet spot" 16.667msecs/frame time limit (msecs).
pageLoadTimeToFrame1: The total time from the beginning of page load until the first application frame rendering is complete (msecs).
pageLoadTimeToFrame10: The total time from the beginning of page load until the 10th application frame has completed rendering (msecs).
timeUntilSmoothFramerate: How long it took to reach a stable animation frame rate (msecs). A stable animation frame rate is reached when 120 subsequent frames are rendered without a single stutter event.

Alternatives

PGO Section

A custom section with PGO information could get added to a WebAssembly module which contains which functions should get compiled eagerly and which functions can get compiled lazily. Such a custom section allows to achieve the advantages of eager compilation (e.g. compiling startup functions already while downloading the WebAssembly module) and the low memory consumption and CPU usage of lazy compilation.
The advantage of such a custom section is that incorrect information in the section would at most result in a performance regression, it would not have an effect on the correctness of the compiled code. Additionally, the section could also just get ignored.
The downside of such a PGO section is that there would probably still be many WebAssembly modules without such a section. For modules without the PGO section we would still have to decide if we want eager or lazy compilation to be the default.
One challenge for a PGO section is standardization. Even though any kind of custom section can be added to a WebAssembly module (e.g. the existing name section), and therefore there could be a Chrome-specific PGO section, it would be better to have agreement with other browser vendors on the specification of the PGO section. However, standardizing the PGO section may delay it's deployment.

Other memory-saving options

Code aging reduces memory consumption of WebAssembly code, both as an alternative to lazy compilation, and as an addition to lazy compilation. Code aging means to deallocate code that we don't expect to get executed again. Code aging would not reduce the memory consumption of WebAssembly during startup, but it would allow reusing this memory later. An impactful first step would be to deallocate all functions after startup that have not been executed so far, or that have not been executed more than once. The deallocated functions would get compiled lazily if they get called after deallocation.
A downside of code aging is that it is not clear if we can reuse the address space of deallocated functions. A dangling pointer to the deallocated function could be a security risk. In that sense it would be better not to compile functions in the first place.

Areas of improvement for lazy compilation

The implementation of lazy compilation is already used in production for asm.js code. However, not much effort has been made to optimize lazy compilation for WebAssembly.

Code publishing

In traces it seems that publishing code is slow, i.e. the writing of generated code into the WebAssembly code space. Publishing code has been optimized so far for eager compilation. It may be worth optimizing publishing code for lazy compilation. The advantage of lazy compilation is that all compilation happens on the main thread. This aspect of lazy compilation may not be true though in multi-threaded WebAssembly code.

Compilation bundles

Functions typically don't just get executed by themselves, they also call other functions. With lazy compilation, these other functions then again have to get compiled on the main thread. We call all functions that get executed together a compilation bundle, as they all get compiled at the same time through lazy compilation. If we could identify such compilation bundles ahead of time, we could compile all functions in the same bundle in parallel when the first function of the bundle gets called.
The advantage of compilation bundles is that code can get compiled on background threads and not on the main thread. Big compilation bundles are expected to be one of the main sources of jank, because all functions in the bundle probably need to be compiled within the same frame.
The downside of this idea is that it is unclear how compilation bundles could get identified, and if the identification of compilation bundles would even be cheaper than Liftoff compilation itself.

Metrics

The main metric to improve would be memory consumption. Additionally CPU time and thereby battery life would improve significantly.

The main metric to improve is the private memory footprint:

- In UMA: Memory.Renderer.PrivateMemoryFootprint
- In telemetry: memory:chrome:renderer_processes:reported_by_os:private_footprint_size

Other interesting memory metrics in telemetry:

- memory:chrome:renderer_processes:reported_by_chrome:v8:allocated_by_malloc
- memory:chrome:renderer_processes:reported_by_chrome:v8:heap