DOI:10.1145/2948991

Article development led by acmqueue
queue.acm.org

Taking advantage of idleness to reduce dropped frames and memory consumption.

BY ULAN DEGENBAEV, JOCHEN EISINGER, MANFRED ERNST, ROSS MCILROY, AND HANNES PAYER

Idle-Time Garbage-Collection Scheduling

GOOGLE'S CHROME WEB browser strives to deliver a smooth user experience. An animation will update the screen at 60FPS (frames per second), giving Chrome approximately 16.6 milliseconds to perform the update. Within these 16.6ms, all input events have to be processed, all animations have to be performed, and finally the frame has to be rendered. A missed deadline will result in dropped frames. These are visible to the user and degrade the user experience. Such sporadic animation artifacts are referred to here as *jank*.[3]

JavaScript, the lingua franca of the Web, is typically used to animate Web pages. It is a garbage-collected programming language where the application developer does not have to worry about memory management. The garbage collector interrupts the application to pass over the memory allocated by the application, determine live memory, free dead memory, and compact memory by moving objects closer together. While some of these garbage-collection phases can be performed in parallel or concurrently to the application, others cannot, and as a result they may cause application pauses at unpredictable times. Such pauses may result in user-visible jank or dropped frames; therefore, we go to great lengths to avoid such pauses when animating Web pages in Chrome.

This article describes an approach implemented in the JavaScript engine V8 used by Chrome to schedule garbage-collection pauses during times when Chrome is idle.[1] This approach can reduce user-visible jank on real-world Web pages and results in fewer dropped frames.

Garbage Collection in V8

Garbage-collector implementations typically optimize for the *weak generational hypothesis*,[6] which states that most of the allocated objects in applications die young. If the hypothesis holds, garbage collection is efficient and pause times are low. If it does not hold, pause times may lengthen.

V8 uses a generational garbage collector, with the JavaScript heap split into a small young generation for newly allocated objects and a large old generation for long-living objects. Since most objects typically die young, this generational strategy enables the garbage collector to perform regular, short garbage collections in the small young generation, without having to trace objects in the large old generation.

The young generation uses a semi-space allocation strategy, where new objects are initially allocated in the young generation's active semi-space. Once a semi-space becomes full, a scavenge operation will trace through the live objects and move them to the other semi-space.

Such a semi-space scavenge is a *minor garbage collection*. Objects that have already been moved in the young generation are promoted to the old generation. After the live objects have been moved, the new semi-space becomes active and any remaining dead objects in the old semi-space are discarded without iterating over them.

The duration of a minor garbage collection therefore depends on the size of the live objects in the young generation. A minor garbage collection is typically fast, taking no longer than one millisecond when most of the objects become unreachable in the young generation. If most objects survive, however, the duration of a minor garbage collection may be significantly longer.

A major garbage collection of the whole heap is performed when the size of live objects in the old generation grows beyond a heuristically derived memory limit of allocated objects. The old generation uses a mark-and-sweep collector with compaction. Marking work depends on the number of live objects that have to be marked, with marking of the whole heap potentially taking more than 100ms for large Web pages with many live objects.

To avoid such long pauses, V8 marks live objects incrementally in many small steps, pausing only the main thread during these marking steps. When incremental marking is completed the main thread is paused to finalize this major collection. First, free memory is made available for the application again by sweeping the whole old-generation memory, which is performed concurrently by dedicated sweeper threads. Afterward, the young generation is evacuated, since we mark through the young generation and have liveness information. Then memory compaction is performed to reduce memory fragmentation in old-generation pages. Young-generation evacuation and old-generation compaction are performed by parallel compaction threads. After that, the object pointers to moved objects in the remembered sets are updated in parallel. All these finalization tasks occur in

a single atomic pause that can easily take several milliseconds.

The Two Deadly Sins of Garbage Collection

The garbage-collection phases outlined here can occur at unpredictable times, potentially leading to application pauses that impact the user experience. Hence, developers often become creative in attempting to sidestep these interruptions if the performance of their application suffers. Here, we look at two controversial approaches that are often proposed and outline their potential problems. These are the two deadly sins of garbage collection.

Sin One: Turning off the garbage collector. Developers often ask for an API to turn off the garbage collector during a time-critical application phase where a garbage-collection pause could result in missed frames. Using such an API, however, complicates application logic and leads to it becoming more difficult to maintain. Forgetting to turn on the garbage collector on a single branch in the program may result in out-of-memory errors. Furthermore, this also complicates the garbage-collector implementation, since it has to support a never-fail allocation mode and must tailor its heuristics to take into account these non-garbage-collecting time periods.

Sin Two: Explicit garbage-collection invocation. JavaScript does not have a Java-style System.gc() API, but some developers would like to have that. Their motivation is proactively to invoke garbage collection during a non-time-critical phase in order to avoid it later when timing is critical. The application, however, has no idea how long such a garbage collection will take and therefore may by itself introduce jank. Moreover, garbage-collection heuristics may get confused if developers invoke the garbage collector at arbitrary points in time.

Given the potential for developers to trigger unexpected side effects with these approaches, they should not interfere with garbage collection. Instead, the runtime system should endeavor to avoid the need for such tricks by providing high-performance application throughput and low-latency pauses during mainline application execution, while scheduling longer-running work during periods of idleness such that it does not impact application performance.

Idle-Task Scheduling

To schedule long-running garbage collection tasks while Chrome is idle, V8 uses Chrome's task scheduler. This scheduler dynamically reprioritizes tasks based on signals it receives from a variety of other components of Chrome and various heuristics aimed at estimating user intent. For example, if the user touches the screen, the scheduler will prioritize screen rendering and input tasks for a period of 100ms to ensure the user interface remains responsive while the user interacts with the Web page.

The scheduler's combined knowledge of task queue occupancy, as well as signals it receives from other components of Chrome, enables it to estimate when Chrome is idle and how long it is likely to remain so. This knowledge is used to schedule low-priority tasks, hereafter called *idle tasks*, which are run only when there is nothing more important to do.

To ensure these idle tasks don't cause jank, they are eligible to run only in the time periods between the current frame having been drawn to screen and the time when the next frame is expected to start being drawn. For example, during active animations or scrolling (see Figure 1), the scheduler uses signals from Chrome's compositor subsystem to estimate when work has been completed for the current frame and what the estimated start time for the next frame is, based on the expected interframe interval (for example, if rendering at 60FPS, the interframe interval is 16.6ms). If no active updates are being made to the screen, the scheduler will initiate a longer idle period, which lasts until the time of the next pending delayed task, with a cap of 50ms to ensure Chrome remains responsive to unexpected user input.

To ensure idle tasks do not overrun an idle period, the scheduler passes a deadline to the idle task when it starts, specifying the end of the current idle

Figure 1. Idle period example.

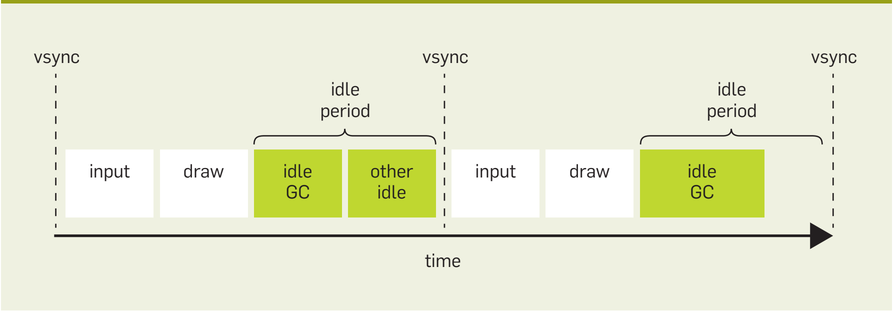
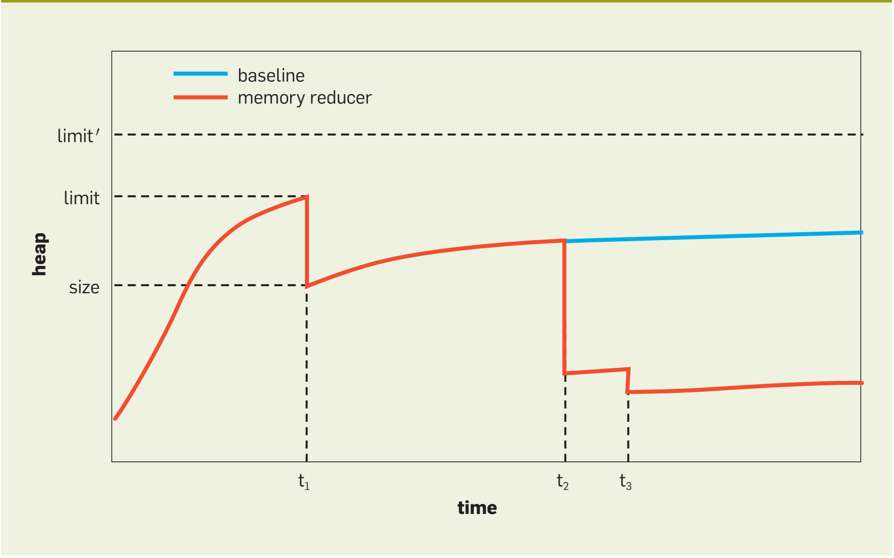


Figure 2. Effect of memory reducer on heap size.



period. Idle tasks are expected to finish before this deadline, either by adapting the amount of work they do to fit within this deadline or, if they cannot complete any useful work within the deadline, by reposting themselves to be executed during a future idle period. As long as idle tasks finish before the deadline, they do not cause jank in Web page rendering.

Idle-Time Garbage-Collection Scheduling in V8

Chrome's task scheduler allows V8 to reduce both jank and memory usage by scheduling garbage-collection work as idle tasks. To do so, however, the garbage collector needs to estimate both when to trigger idle-time garbage-collection tasks and how long those tasks are expected to take. This allows the garbage collector to make the best use of the available idle time without going past an idle-tasks deadline. This section describes implementation details of idle-time scheduling for minor and major garbage collections.

Minor garbage-collection idle-time scheduling. Minor garbage collection cannot be divided into smaller work chunks and must be performed either completely or not at all. Performing minor garbage collections during idle time can reduce jank; however, being too proactive in scheduling a minor garbage collection can result in promotion of objects that could otherwise die in a subsequent non-idle minor garbage collection. This could increase the old-generation size and the latency of future major garbage collections. Thus, the heuristic for scheduling minor garbage collections during idle time should balance between starting a garbage collection early enough that the young-generation size is small enough to be collectable during regular idle time, and deferring it long enough to avoid false promotion of objects.

Whenever Chrome's task scheduler schedules a minor garbage-collection task during idle time, V8 estimates if the time to perform the minor garbage collection will fit within the idle-task deadline. The time estimate is computed using the average garbage-collection speed and the current size of the young generation. It also estimates the young-generation growth rate and performs an idle-time minor garbage collection only if the estimate is that at the next idle period the size of the young generation is expected to exceed the size that could be collected within an average idle period.

Chrome's task scheduler allows V8 to reduce both jank and memory usage by scheduling garbage-collection work as idle tasks.

Major garbage-collection idle-time scheduling. A major garbage collection consists of three parts: initiation of incremental marking, several incremental marking steps, and finalization. Incremental marking starts when the size of the heap reaches a certain limit, configured by a heap-growing strategy. This limit is set at the end of the previous major garbage collection, based on the heap-growing factor f and the total size of live objects in the old generation: limit = $f \cdot$ size.

As soon as an incremental major garbage collection is started, V8 posts an idle task to Chrome's task scheduler, which will perform incremental marking steps. These steps can be linearly scaled by the number of bytes that should be marked. Based on the average measured marking speed, the idle task tries to fit as much marking work as possible into the given idle time. The idle task keeps reposting itself until all live objects are marked. V8 then posts an idle task for finalizing the major garbage collection. Since finalization is an atomic operation, it is performed only if it is estimated to fit within the allotted idle time of the task; otherwise, V8 reposts that task to be run at a future idle time with a longer deadline.

Memory reducer. Scheduling a major garbage collection based on the allocation limit works well when the Web page shows a steady allocation rate. If the Web page becomes inactive and stops allocating just before hitting the allocation limit, however, there will be no major garbage collection for the whole period while the page is inactive. Interestingly, this is an execution pattern that can be observed in the wild. Many Web pages exhibit a high allocation rate during page load as they initialize their internal data structures. Shortly after loading (a few seconds or minutes), the Web page often becomes inactive, resulting in a decreased allocation rate and decreased execution of JavaScript code. Thus, the Web page will retain more memory than it actually needs while it is inactive.

A controller, called *memory reducer*, tries to detect when the Web page becomes inactive and proactively schedules a major garbage collection even if the allocation limit is not reached. Figure 2 shows an example of major garbage-collection scheduling.

The first garbage collection happens at time t_1 because the allocation limit is reached. V8 sets the next allocation limit based on the heap size. The subsequent garbage collections at times t_2 and t_3 are triggered by the memory reducer before limit is reached. The dotted line shows what the heap size would be without the memory reducer.

Since this can increase latency, Google developed heuristics that rely not only on the idle time provided by Chrome's task scheduler, but also on whether the Web page is now inactive. The memory reducer uses the JavaScript invocation and allocation rates as signals for whether the Web page is active or not. When the rate drops below a predefined threshold, the Web page is considered to be inactive and major garbage collection is performed in idle time.

Silky Smooth Performance

Our aim with this work was to improve the quality of user experience for animation-based applications by reducing jank caused by garbage collection. The quality of the user experience for animation-based applications depends not only on the average frame rate, but also on its regularity. A variety of metrics have been proposed in the past to quantify the phenomenon of jank—for example, measuring how often the frame rate has changed, calculating the variance of the frame durations, or simply using the largest frame duration. Although these metrics provide useful information, they all fail to measure certain types of irregularities. Metrics that are based on the distribution of frame durations, such as variance or largest frame duration, cannot take the temporal order of frames into account. For example, they cannot distinguish between the case where two dropped frames are close together and the case where they are further apart. The former case is arguably worse.

We propose a new metric to overcome these limitations. It is based on the discrepancy of the sequence of frame durations. Discrepancy is traditionally used to measure the quality of samples for Monte Carlo integration. It quantifies how much a sequence of numbers deviates from a uniformly distributed sequence. Intuitively, it measures the duration of the worst jank. If only a single frame is dropped, the discrepancy metric is equal to the size of the gap between the drawn frames. If multiple frames are dropped in a row—with some good frames in between—the discrepancy will report the duration of the entire region of bad performance, adjusted by the good frames.

Discrepancy is a great metric for quantifying the worst-case performance of animated content. Given the timestamps when frames were drawn, the discrepancy can be computed in $O(N)$ time using a variant of Kadane's algorithm for the maximum subarray problem.

The online Web Graphics Library (WebGL) benchmark OortOnline (http://oortonline.gl/#run) demonstrates jank improvements of idle-time garbage-collection scheduling. Figure 3 shows these improvements: frame-time discrepancy, frame time, number of frames missed because of garbage collection, and total garbage-collection time compared with the baseline on the oortonline.gl benchmark.

Frame-time discrepancy is reduced on average from 212ms to 138ms. The average frame-time improvement is from 17.92ms to 17.6ms. We observed that 85% of garbage-collection work was scheduled during idle time, which significantly reduced the amount of garbage-collection work performed during time-critical phases. Idle-time garbage-collection scheduling increased the

Figure 3. Improvements to the OortOnline.gl benchmark.

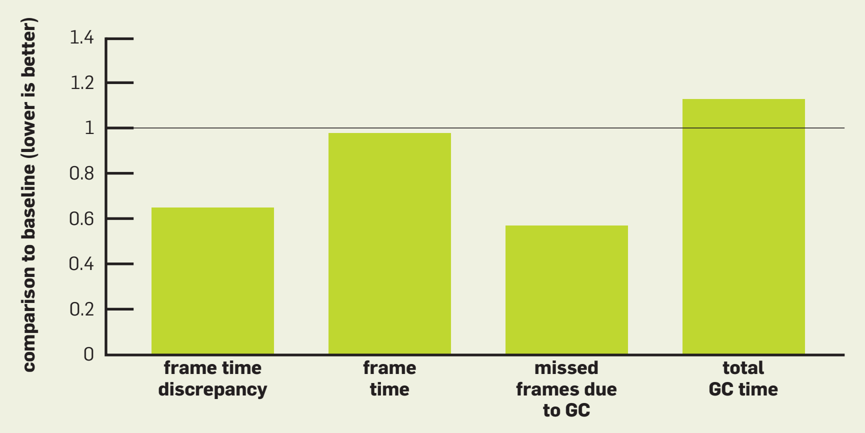
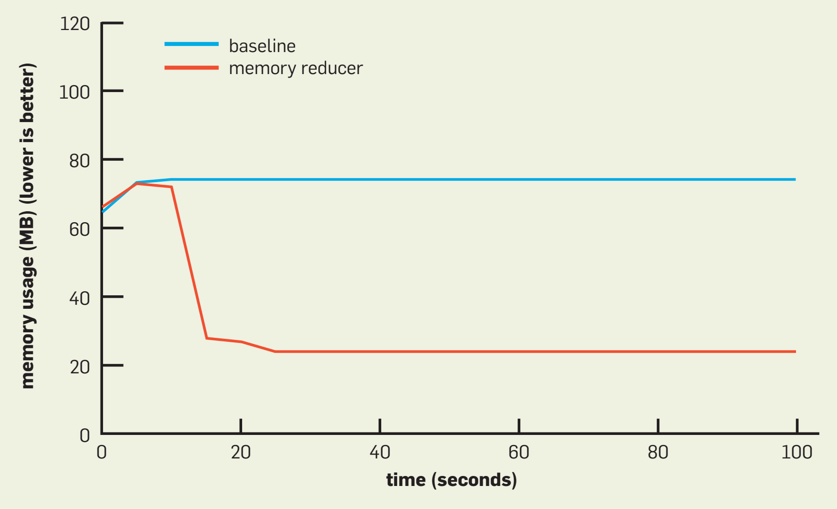


Figure 4. Memory usage comparison.



total garbage-collection time by 13% to 780ms. This is because scheduling garbage collection proactively and making faster incremental marking progress with idle tasks resulted in more garbage collections.

Idle-time garbage collection also improves regular Web browsing. While scrolling popular Web pages such as Facebook and Twitter, we observed that about 70% of the total garbage-collection work is performed during idle time.

The memory reducer kicks in when Web pages become inactive. Figure 4 shows an example run of Chrome with and without the memory reducer on the Google Web Search page. In the first few seconds both versions use the same amount of memory as the Web page loads and allocation rate is high. After a while the Web page becomes inactive since the page has loaded and there is no user interaction. Once the memory reducer detects that the page is inactive, it starts a major garbage collection. At that point the graphs for the baseline and the memory reducer diverge. After the Web page becomes inactive, the memory usage of Chrome with the memory reducer decreases to 34% of the baseline.

A detailed description of how to run the experiments presented here to reproduce these results can be found in the 2016 Programming Language Design and Implementation (PLDI) artifact evaluation document.[2]

Other Idle-Time Garbage-Collected Systems

A comprehensive overview of garbage collectors taking advantage of idle times is available in a previous article.[4] The authors classify different approaches in three categories: slack-based systems where the garbage collector is run when no other task in the system is active; periodic systems where the garbage collector is run at predefined time intervals for a given duration; and hybrid systems taking advantage of both ideas. The authors found that, on average, hybrid systems provide the best performance, but some applications favor a slack-based or periodic system.

Our approach of idle-time garbage-collection scheduling is different. Its main contribution is that it profiles the application and garbage-collection components to predict how long garbage-collection operations will take and when the next minor or major collection will occur as a result of application allocation throughput. That information allows efficient scheduling of garbage-collection operations during idle times to reduce jank while providing high throughput.

Concurrent, Parallel, Incremental Garbage Collection

An orthogonal approach to avoid garbage-collection pauses while executing an application is achieved by making garbage-collection operations concurrent, parallel, or incremental. Making the marking phase or the compaction phase concurrent or incremental typically requires read or write barriers to ensure a consistent heap state. Application throughput may degrade because of expensive barrier overhead and code complexity of the virtual machine.

Idle-time garbage-collection scheduling can be combined with concurrent, parallel, and incremental garbage-collection implementations. For example, V8 implements incremental marking and concurrent sweeping, which may also be performed during idle time to ensure fast progress. Most importantly, costly memory-compaction phases such as young-generation evacuation or old-generation compaction can be efficiently hidden during idle times without introducing costly read or write barrier overheads.

For a best-effort system, where hard realtime deadlines do not have to be met, idle-time garbage-collection scheduling may be a simple approach to provide both high throughput and low jank.

Beyond Garbage Collection and Conclusion

Idle-time garbage-collection scheduling focuses on the user's expectation that a system that renders at 60 frames per second appears silky smooth. As such, our definition of idleness is tightly coupled to on-screen rendering signals. Other applications can also benefit from idle-time garbage-collection scheduling when an appropriate definition of idle time is applied. For example, a node.js-based server that is built on V8 could forward idle-time periods to the V8 garbage collector while it waits for a network connection.

The use of idle time is not limited to garbage collection. It has been exposed to the Web platform in the form of the requestIdleCallback API,[5] enabling Web pages to schedule their own callbacks to be run during idle time. As future work, other management tasks of the JavaScript engine could be executed during idle time (for example, compiling code with the optimizing just-in-time compiler that would otherwise be performed during JavaScript execution). C

Related articles on queue.acm.org

Real-time Garbage Collection
David F. Bacon
http://queue.acm.org/detail.cfm?id=1217268

A Conversation with David Anderson
http://queue.acm.org/detail.cfm?id=1080872

Network Virtualization: Breaking the Performance Barrier
Scott Rixner
http://queue.acm.org/detail.cfm?id=1348592

References

1. Degenbaev, U., Eisinger, J., Ernst, M., McIlroy, R., Payer, H. Idle time garbage collection scheduling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (2016).
2. Degenbaev, U., Eisinger, J., Ernst, M., McIlroy, R., Payer, H. PLDI'16 Artifact: Idle time garbage collection scheduling (Santa Barbara, CA, June 13-17, 2016) 570–583. ACM, 978-1-4503-4261-2/16/06; https://goo.gl/AxvigS.
3. Google Inc. The RAIL performance model; http://developers.google.com/Web/tools/chrome-devtools/profile/evaluate-performance/rail.
4. Kalibera, T., Pizlo, F., Hosking, A. L., Vitek, J. Scheduling real-time garbage collection on uniprocessors. *ACM Trans. Computer Systems 29*, 3 (2011), 8:1–8:29.
5. McIlroy. R. Cooperative scheduling of background tasks. W3C editor's draft, (2016); https://w3c.github.io/requestidlecallback/.
6. Ungar, D. 1984. Generation scavenging: a nondisruptive high-performance storage reclamation algorithm. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (SDE 1).

Ulan Degenbaev is a software engineer at Google, working on the garbage collector of the V8 JavaScript engine.

Jochen Eisinger is a software engineer at Google, working on the V8 JavaScript engine and Chrome security. Prior to that, he worked on various other parts of Chrome.

Manfred Ernst is a software engineer at Google, where he works on virtual reality. Prior to that, he integrated a GPU rasterization engine into the Chrome Web browser. Ernst was also research scientist at Intel Labs and a cofounder and the CEO of Bytes+Lights.

Ross McIlroy is a software engineer at Google and tech lead of V8's interpreter effort. He previously worked on Chrome's scheduling subsystem and mobile optimization efforts. Previously, McIlroy worked on various operating-system and virtual-machine research projects, including Singularity, Helios, Barrelfish, and HeraJVM.

Hannes Payer is a software engineer at Google, tech lead of the V8 JavaScript garbage collection effort, and a virtual-machine enthusiast. Prior to V8, Payer worked on Google's Dart virtual machine and various Java virtual machines.