Idle Time Garbage Collection Scheduling

Ulan Degenbaev[+] Jochen Eisinger[+] Manfred Ernst[#] Ross McIlroy[*] Hannes Payer[+]

Google Germany[+], UK[*], USA[#]

{ulan,eisinger,ernstm,rmcilroy,hpayer}@google.com

Abstract

Efficient garbage collection is increasingly important in today's managed language runtime systems that demand low latency, low memory consumption, and high throughput. Garbage collection may pause the application for many milliseconds to identify live memory, free unused memory, and compact fragmented regions of memory, even when employing concurrent garbage collection. In animation-based applications that require 60 frames per second, these pause times may be observable, degrading user experience. This paper introduces idle time garbage collection scheduling to increase the responsiveness of applications by hiding expensive garbage collection operations inside of small, otherwise unused idle portions of the application's execution, resulting in smoother animations. Additionally we take advantage of idleness to reduce memory consumption while allowing higher memory use when high throughput is required. We implemented idle time garbage collection scheduling in V8, an open-source, production JavaScript virtual machine running within Chrome. We present performance results on various benchmarks running popular webpages and show that idle time garbage collection scheduling can significantly improve latency and memory consumption. Furthermore, we introduce a new metric called *frame time discrepancy* to quantify the quality of the user experience and precisely measure the improvements that idle time garbage collection scheduling provides for a WebGL-based game benchmark. Idle time garbage collection scheduling is shipped and enabled by default in Chrome.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors - Memory management (garbage collection); D4 [*Process Management*]: Scheduling

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908106

General Terms Algorithms, Languages, Measurement, Performance

Keywords Garbage Collection, Memory Management, Virtual Machines, Scheduling, JavaScript, Web Applications, Browser Technology

1. Introduction

Many modern language runtime systems such as Chrome's V8 JavaScript engine dynamically manage memory for running applications so that developers do not need to worry about it themselves. The engine periodically passes over the memory allocated to the application, determines live memory, and frees dead memory. This process is known as garbage collection.

Chrome strives to deliver a smooth user experience rendering the display at 60 frames per second (FPS). Although V8 already attempts to perform garbage collection both incrementally in small chunks and perform some operations concurrently on multiple threads, larger garbage collection operations can and do occur at unpredictable times – sometimes in the middle of an animation – pausing execution and preventing Chrome from achieving the 60 FPS goal.

Chrome 41 includes a task scheduler for the Blink rendering engine which enables prioritization of latency-sensitive tasks to ensure the browser remains responsive and snappy. As well as being able to prioritize work, this task scheduler has centralized knowledge of how busy the system is, what tasks need to be performed, and how urgent each of these tasks are. The scheduler can estimate when Chrome is likely to be idle and approximately how long it expects to remain idle.

An example of this occurs when Chrome is showing an animation on a webpage. The animation will update the screen at 60 FPS, giving Chrome around 16.6 ms of time to perform the update. Chrome will start work on the next frame as soon as the current frame has been displayed, performing input, animation and frame rendering tasks for this new frame. If Chrome completes all this work in less than 16.6 ms, then it has nothing else to do for the remaining time until it must start rendering the next frame. Chrome's scheduler enables V8 to take advantage of this idle time period by

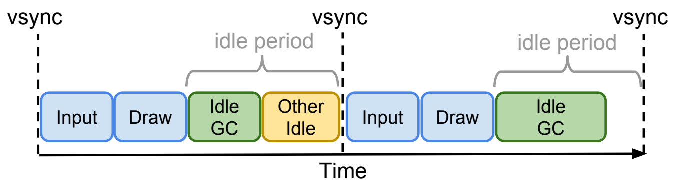


Figure 1: Idle times during rendering of an animation.

scheduling special idle tasks which run when Chrome would otherwise be idle.

Idle tasks are given a deadline which is the scheduler's estimate of how long it expects to remain idle. In the animation example depicted in Figure 1, this is the time at which the next frame should start being drawn. In other situations (e.g., when no on-screen activity is happening) this could be the time when the next pending task is scheduled to be run. The deadline is used by the idle task to estimate how much work it can do without causing jank[1] or delays in input response.

There remain three key challenges for the garbage collector to efficiently take advantage of idle tasks. First, which garbage collection operation should be scheduled during idle time? Second, how long will a given garbage collection operation take? Third, when should a given garbage collection operation be scheduled proactively? If the right garbage collection operation is scheduled at the right time, garbage collection interruptions may be hidden from latency-sensitive tasks allowing 60 FPS.

Summary of Contributions

- An implementation of idle tasks in the Blink scheduler;
- A garbage collection performance profiler in V8 which allows V8 to estimate duration of future garbage collection operations to schedule them during application idle times based on various heuristics;
- Frame time discrepancy, a novel metric to quantify user experience for animations;
- A set of novel real-world benchmarks together with a thorough evaluation.

The rest of the paper is structured as follows. In Section 2 we introduce Chrome and its task scheduling, describe how garbage collection works in the V8 JavaScript virtual machine, and discuss Chrome's benchmarking framework *Telemetry*. Section 3 discusses metrics to quantify user experience and introduces frame time discrepancy, a novel metric to quantify jank for animation-based applications. Section 4 introduces idle tasks and describes the algorithm used by the Blink scheduler to estimate periods where Chrome is likely to remain idle, such that idle tasks can execute without introducing jank. Section 5 introduces idle time garbage collection scheduling with its heuristics to reduce jank and memory consumption. Section 6 discusses related work. Section 7 demonstrates that idle time garbage collection scheduling can reduce garbage collection latency and memory consumption on various real-world webpages followed by conclusions in Section 8.

[1] Latency introduced by e.g. the garbage collector causing observable sporadic animation artifacts due to dropped frames. Also referred to as jitter or hiccups.

2. Background

In this section we introduce all the necessary ingredients for idle time garbage collection scheduling. We give a brief overview about rendering of webpages in Chrome, task scheduling in the Blink scheduler, V8 garbage collection, and the Telemetry benchmarking framework.

2.1 Webpage Rendering

Webpages are rendered by Chrome in a multi-step process. After being parsed, the page's HTML, JavaScript and CSS files are used to build up a Document Object Model (DOM) tree. This DOM tree represents the internal structure of the page and can be manipulated by the page via JavaScript in order to change the visible content of the webpage dynamically. Chrome uses this DOM tree to build an internal render tree, representing the operations required to display the page elements on screen during rendering. If the page modifies its DOM tree, Chrome performs a re-layout operation to discover how the render tree should be modified in order to reflect these changes. If a modification to the render tree causes a change to the content displayed on screen, or the user performs a scroll which changes what content is displayed, Chrome's compositor will draw a new frame to the display.

Chrome's compositor is multi-threaded, with the main thread (where JavaScript functions are executed) responsible for maintaining the DOM and renderer tree, and the compositor thread responsible for drawing the render tree to screen. When the compositor decides to draw a frame, it signals to the main thread that it is beginning a new frame. When the main thread receives this signal, it performs any JavaScript operations required for the new frame (e.g., running requestAnimationFrame callbacks as well as batching up input operations and calling any registered JavaScript input handlers). It then updates the render tree in response to any changes performed to the DOM tree. Once the render tree is updated, the main thread will acknowledge that it has committed its processing for the frame, and the compositor thread will draw the frame to the display.

Certain operations which do not involve JavaScript callbacks (e.g. scrolling) can happen entirely on the compositor thread, thereby avoiding being bottlenecked by other operations happening on the main thread. However, many animations require interaction with the main thread and thus can be delayed if the main thread is busy, for example, due to a garbage collection operation by V8.

2.2 Task Scheduling

Chrome employs a task scheduler [26] on the main thread in order to reduce the likelihood of frame updates being delayed by long running operations on the main thread. Tasks are posted to type-specific queues such as queues for compositing operations, input handling tasks, JavaScript timer execution tasks, and page loading tasks. Tasks on the same queue are run in the same order as they were posted, to maintain task ordering semantics required by Chrome, however tasks posted on different queues are free to be reordered by the scheduler.

The scheduler dynamically re-prioritizes the task queues based on signals it receives from a variety of other components of Chrome and various heuristics aimed at estimating user intent. For example, while the page is loading, tasks on the loading task queue, such as HTML parsing or network requests, are given priority. Alternatively, if a touch event is detected, the scheduler will prioritize compositing and input tasks for a period of 100ms, on the assumption that a follow-up input events is likely to occur within this time interval as the user interacts with the webpage by scrolling, tapping or clicking.

The scheduler's combined knowledge of task queue occupancy and signals from other components of Chrome enables it to estimate when it is idle and how long it is likely to remain idle. This knowledge is used by the scheduler to schedule low-priority tasks which are only run when there is nothing more important to do. These low-priority tasks, hereafter called *idle tasks*, are put on a queue which is only run when all other queues are empty, and executed for a limited time to ensure latency sensitive tasks are not impacted by their execution.

Section 4 provides more details on our approach to schedule idle tasks, and Section 5 details how idle tasks are employed by V8's garbage collector to reduce user-perceived jank.

2.3 V8 Garbage Collection

V8 is an industrial-strength virtual machine for JavaScript that can be embedded into a C/C++ application through a set of programmable APIs. V8 is most prominently used by both the Chrome and Opera web browsers and the server-side node.js framework [33].

V8 uses a generational garbage collector with the JavaScript heap split into a small young generation (up to 16M) for newly allocated objects and a large old generation (up to 1.4G) for long living objects. Since most objects typically die young in regular webpages [1], 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. Moreover, when V8 detects that the generational hypothesis does not hold for some allocation sites, it performs dynamic allocation-site-based pretenuring [14] to allocate long living objects directly in the old generation.

Both, the young and old generations are organized in pages of 1M in size, which allows efficient heap growing and shrinking strategies. Objects larger than 600K are allocated on separate, arbitrary size pages which are considered to be part of the old generation. The old generation also includes the code space with all executable code objects and the map space with the evolved hidden classes [18]. Note that allocations from JavaScript do not require synchronization since JavaScript is single-threaded and each JavaScript context gets its own private heap.

The young generation uses a semi-space allocation strategy, where new objects are initially allocated in the young generation's active semi-space using bump-pointer allocation, which can be inlined in generated code [13]. Once a semi-space becomes full, a scavenge operation will trace through the live objects and move them to the other semi-space. We refer to such a semi-space scavenge as a minor garbage collection. Objects which have been moved already once 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 live objects in the young generation. A scavenge will be fast (<1 ms) when most of the objects become unreachable in the young generation. However, if most objects survive a scavenge, the duration of the scavenge may be significantly longer.

The old generation uses bump pointer allocation on the fast path, which can be inlined in generated code, for pretenured allocations. Segregated free-lists are used to refill the bump-pointer. 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 limit. The old generation uses a mark-sweep-compact collector with several optimizations to improve latency and memory consumption. Marking latency depends on the number of live objects that have to be marked, with marking of the whole heap potentially taking more than 100 ms for large webpages. In order to avoid pausing the main thread for such long periods, V8 marks live objects incrementally in many small steps, with the aim to keep each marking step below 5 ms in duration. A Dijkstra-style write barrier is used to maintain a tricolor marking scheme.

After marking, the free memory is made available for the application again by sweeping the whole old generation memory. This task is performed concurrently by dedicated sweeper threads. The main thread only contributes to sweeping if the sweeper threads are not making progress. Afterwards, we perform a young generation evacuation, since we mark through the young generation and have liveness information. Similarly to a minor collection, live objects are moved to the other semi-space or, in the case of already copied live objects, are moved to the old generation. Then memory compaction is performed to reduce memory fragmentation in the old generation. Finally, the object pointers to moved objects in the remembered sets are updated sequentially.

The duration of a major collection is essentially linear in the used heap size. V8 tries to keep these pauses short (<6 ms) to ensure smooth animations [20].

2.4 Measuring Performance using Telemetry

Telemetry is Chrome's performance testing framework, which enables the creation of benchmarks which automatically perform various configurable actions on a set of webpages and report a variety of performance metrics. The Telemetry framework is designed to run on all platforms that Chrome supports, and to not require a special build. Instead, Telemetry benchmarks can be run on any Chrome binary. Therefore, Telemetry was designed to run with minimal (ideally no) performance impact on the browser itself. Internally, Telemetry relies on manual instrumentation of Chrome's and V8's source to gather relevant events. It uses the remote inspector protocol of Chrome's built-in developer tools to retrieve this tracing information and control the browser during the benchmark. A Telemetry benchmark consists of a page set which is a collection of HTML, JavaScript, and CSS files that were captured from a real browsing session, a set of browser options, and a so-called measurement. Page sets can have actions associated with them that are executed during the benchmark, such as clicking on a certain element, or performing a swipe gesture. A measurement evaluates Chrome's performance on the given page set. It defines the classes of traces to collect, and how to aggregate these traces into measurable performance metrics.

In the context of this paper, a page set is a selection of popular webpages with an optional configurable scroll gesture or a WebGL benchmark without any particular actions. The respective measurement evaluates how much jank was observed on the page, how long the individual garbage collection tasks were, and whether or not they fell inside idle periods. The experiments presented in Section 7 are performed using the Telemetry framework.

3. Discrepancy as a Metric for Frame Rate Regularity

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 this phenomenon, also referred to as jank, judder, or jitter. 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 those metrics do 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. A sequence of numbers $S = \{s_1, s_2, s_3, \ldots\}$ is uniformly distributed on an interval $[a, b]$, if for any subinterval $[c, d]$:

$$\lim_{n\to\infty} \frac{|S_n \cap [c, d]|}{n} = \frac{d-c}{b-a},$$

where $S_n = \{s_1, s_2, \ldots, s_n\}$. The discrepancy of the sequence is defined as:

$$D_n = sup_{a \le c \le d \le b} \left| \frac{|S_n \cap [c, d]|}{n} - \frac{d-c}{b-a} \right|$$

S is uniformly distributed if $\lim_{n\to\infty} D_n = 0$.

In our case, S is the finite sequence of timestamps when a frame was drawn. We do not use D_n directly, because its value would improve if additional 'good' frames were added to a given sequence. Instead, we calculate the absolute discrepancy $\bar{D_n} = D_n \cdot (b - a)$. It has the additional benefit of having milliseconds as the unit, while the relative discrepancy D_n is unit-less. In some sense it measures the duration of the worst irregularity in frame rate.

We also normalize the samples before calculating the discrepancy. Consider the sequences $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$ with

$$a_i = \frac{i-1}{N-1}, \quad b_i = \frac{i-\frac{1}{2}}{N}, \quad i = 1, \ldots, N.$$

The discrepancy of A is $\frac{2}{N}$, twice the discrepancy of B. In our case we do not want to distinguish between the two cases, as our original domain of the timestamps is not bounded (as it is for samples in Monte Carlo integration). We map the timestamps linearly to the interval $[0, 1]$ such that the first value becomes $\frac{1}{2N}$ and the last value becomes $1 - \frac{1}{2N}$. Discrepancy is then calculated for this normalized sequence.

The properties of the discrepancy metric are easiest understood by looking at the examples in Figure 2.

Each line represents a series of timestamps. Black dots represent frames that were drawn, white dots represent frames that were missed. The spacing between the dots is 1 VSYNC interval, which equals 16.6 ms for a 60 Hz refresh rate. The discrepancies, expressed in VSYNC intervals, are: $\bar{D}(S_1) = 1$, $\bar{D}(S_2) = 2$, $\bar{D}(S_3) = 2$, $\bar{D}(S_4) = \frac{25}{9}$,

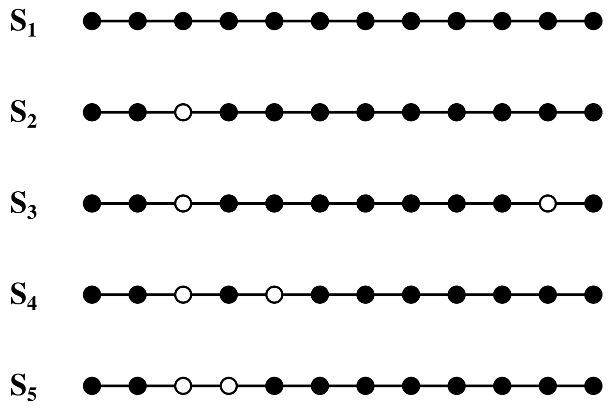


Figure 2: Discrepancy examples.

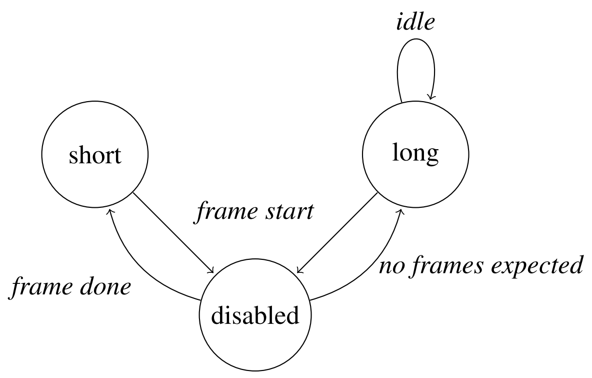


Figure 3: Scheduler's idle period states

$\bar{D}(S_5) = 3$. For the perfect run S_1, the discrepancy is equal to the spacing between the frames. If a single frame is dropped (S_2), the discrepancy is the length of the largest gap between frames. This does not change if an additional frame is dropped far away from the first one (S_3). Remember that the discrepancy measures the worst case performance, not the average. That is why we usually combine it with the mean frame duration to distinguish a single dropped frame from repeated frame drops (which is a worse experience). For the sequence S_4, the discrepancy does increase, because the two dropped frames are now close together; They are treated as a single area of irregular frame rate. The value $\frac{25}{9}$ is computed as follows: The largest discrepancy is between the second and fourth frame. After normalization, these are located at $c = \frac{5}{100} + \frac{9}{110} = \frac{29}{220}$ and $d = \frac{5}{100} + \frac{45}{110} = \frac{101}{220}$. One out of ten samples is located in the open interval (c, d). The discrepancy then becomes $(d - c) - \frac{1}{10} = \frac{25}{110}$ (Note that $b - a = 1$ after normalization). Multiplying by the inverse scale factor of the normalization yields the absolute discrepancy $\bar{D}(S_4) = \frac{25}{110} \cdot \frac{110}{9} = \frac{25}{9}$. The discrepancy of S_5 is even higher, because there is no good frame between the dropped frames. $\bar{D}(S_5) = 3$, the length of the gap. Discrepancy can be computed in $O(N)$ time using a variant of Kadane's algorithm for the maximum subarray problem [8].

Frame time discrepancy works well for animations that require a constant and steady frame rate. We will use a WebGL-based game later in the experiments in Section 7 and show significant improvements on that metric. We will also report improvements on the frames-per-second metric but we will show that the improvement on that metric is less obvious.

4. Idle Task Scheduling

As described in Section 2.2, Chrome's scheduler enables the scheduling of low-priority idle tasks during times when the browser would be otherwise idle. In this section we describe how idle tasks are scheduled to ensure they do not impact latency sensitive tasks on the same thread.

Idle tasks are posted on a queue which has lower priority than all other queues in the scheduler. Tasks on this queue are only eligible to be run during scheduler-defined idle periods. An idle period is defined as the period of time between all work being completed on the main thread to draw a given frame and the time that the next frame is expected to start being drawn.

During active animations or scrolling, the compositor will signal the scheduler every time it starts to draw a frame. This signal also provides the expected inter-frame interval of future frames (e.g., if rendering at 60 FPS, the inter-frame interval will be 16.6 ms). When the main thread finishes it's work for the current frame, the compositor will commit the frame, and will again signal the scheduler. On this signal, the scheduler checks the expected time of the next frame. If this is in the future the scheduler starts an idle period which lasts until the expected time of the next frame (see Figure 1).

If the compositor decides that it no longer needs to draw frames (e.g, due to an animation or scrolling finishing) then it informs the scheduler that no frames are expected. This signal initiates a longer idle period, which lasts until either the time of the next pending delayed task, or 50 ms in the future, whichever is sooner. These longer idle periods are capped at 50 ms to ensure that Chrome remains responsive to unexpected user input, with 50 ms being chosen because studies [28] suggest that a response to user input within 100 ms is perceived as instantaneous by users. Once this longer idle period ends, a new idle period will be started immediately if the browser remains idle. However if the compositor signals the start of a new frame, the scheduler ends the current idle period, even if it has time remaining. Figure 3 shows a simplified state transition diagram of scheduler idle periods.

When an idle period starts, idle tasks become eligible to be run by the scheduler. In order to ensure that idle tasks do not run out-with an idle period, the scheduler passes a deadline to an idle task when it starts, which specifying the end of the current idle 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 the cannot complete any useful work within the deadline, 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 webpage rendering.

As well as being employed by V8's garbage collector, these idle task are exposed to the web platform via the `requestIdleCallback` API [27].

5. Idle Time Garbage Collection Scheduling

In this section we describe how V8 uses idle tasks to reduce latency and memory usage. V8 posts idle tasks to the scheduler in order to perform both minor and major garbage collection during idle time. A minor garbage collection cannot be divided into smaller work chunks and must be either performed completely or not performed at all. A major garbage collection consists of three parts: start of incremental marking, several incremental marking steps, and finalization. These work chunks have different latency / memory trade-offs when performed during idle time.

Although starting incremental marking is a low latency operation, it leads to incremental steps and finalization that can induce long pauses. Thus starting incremental marking can hurt latency but is crucial for reducing memory usage. Performing incremental marking steps and finalization during idle time benefits both latency and memory usage.

The impact of minor garbage collections during idle time depends on the scheduling heuristics and the lifetime of the objects in the young generation. Performing minor garbage collections during idle time usually improves latency and has little effect on memory usage. However, proactive scheduling can result in promotion of objects that would otherwise die later in non-idle minor garbage collection. This would increase old generation size and increase latency of major garbage collections. Thus the heuristic for scheduling minor garbage collections during idle time should balance between starting garbage collection too early, which would unnecessarily promote objects, and too late, which would make the young generation size too large to be collectable during regular idle times. The following subsections describe heuristics for scheduling garbage collection work during idle time.

5.1 Minor Garbage Collection Idle Time Scheduling

In order to implement idle time minor garbage collection we need:

1. a predicate that tells us whether to perform minor garbage collection or not, given the idle task's deadline and the state of the new generation
2. a mechanism to post an idle task on the Blink scheduler that performs a minor garbage collection at an appropriate time

V8 performs idle time minor garbage collection only if its estimated time fits within the given idle task's deadline and there are enough objects allocated in the young generation. Let H be the total size in bytes of objects in the young generation, $\bar{S}$ be the average speed of previously observed minor garbage collection in bytes per second, T be the current idle task deadline in seconds, $\bar{T}$ be the average idle task deadline in seconds. V8 performs minor garbage collection if $max(\bar{T} \cdot \bar{S} - N, H_{min}) < H \leq \bar{S} \cdot T$, where N is the estimated number of bytes that will be allocated before the next idle task and H_{min} is the minimum young generation size that warrants garbage collection. The $\bar{T} \cdot \bar{S} - N < H$ condition can be rewritten as $\bar{T} \cdot \bar{S} < H + N$, which estimates if the next idle time minor garbage collection would overrun the average idle task deadline. If it does then V8 needs to perform idle time minor garbage collection in the current idle task.

In order to request idle time for minor garbage collection from the Blink scheduler, V8 puts allocation bailout markers at every N bytes of the new generation. When an allocation from the JavaScript code crosses an allocation bailout marker, the allocation takes the slow path and invokes a V8 runtime function that posts an idle task to the Blink scheduler. We found that $N = 512KB$ works well in practice, this number is large enough that allocation bailouts do not reduce throughput and it is small enough that V8 has multiple opportunities to post idle tasks.

5.2 Major Garbage Collection Idle Time Scheduling

In this subsection we deal with incremental marking steps and finalization of major garbage collection during idle time assuming that incremental marking is already started. We postpone description of heuristics for starting incremental marking until the next subsection.

As soon as incremental major garbage collection (idle or non-idle) is started, V8 posts an idle task to the Blink scheduler. The callback of the idle task performs incremental marking steps. The 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. Let t_{idle} be the deadline in seconds of the idle task, M be the marking speed in byte per second, then $\lfloor t_{idle} \cdot M \rfloor$ bytes will be marked in the idle task.

The idle task keeps reposting itself until the whole heap is marked. After that V8 posts an idle task for finalizing major garbage collection. The finalization is performed only if its estimated time fits the allotted idle time. V8 estimates the time using the measured speed of previous finalizations.

5.3 Memory Reducer

This section describes the *memory reducer*, a controller for scheduling major garbage collections which tries to reclaim memory for inactive webpages during idle time. Before we go into details about starting major garbage collections dur-

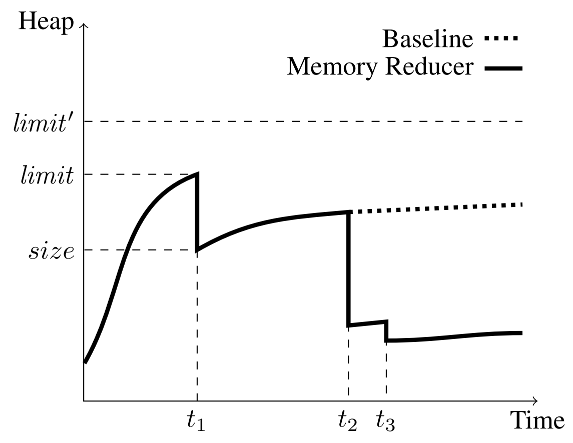
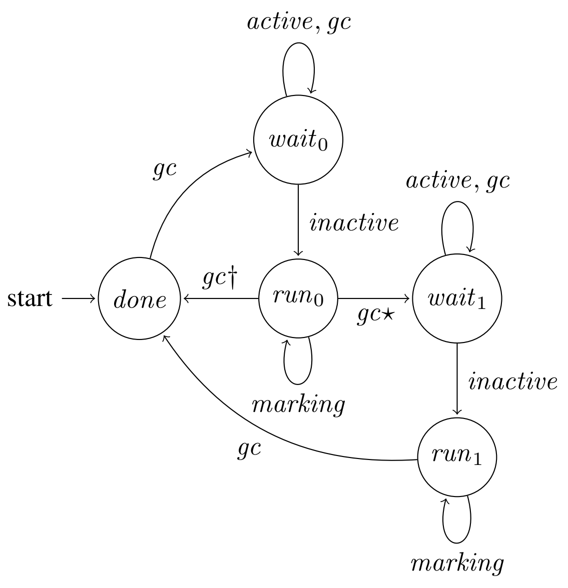


Figure 4: Effect of memory reducer on heap size of an inactive webpage.



$$\begin{tikzcd}
 & wait_0 \arrow[loop above, "{active, gc}"] \arrow[d, "inactive"] & \\
\text{start} \arrow[r] & done \arrow[u, bend left, "gc"] & run_0 \arrow[l, "gc\dagger"'] \arrow[r, "gc\star"'] \arrow[loop below, "marking"] & wait_1 \arrow[loop above, "{active, gc}"] \arrow[d, "inactive"] \\
 & & & run_1 \arrow[ulll, bend left, "gc"] \arrow[loop below, "marking"]
\end{tikzcd}$$

Figure 5: Memory reducer states

ing idle time, let us review how non-idle major garbage collections are triggered in V8. Major garbage collections are typically performed 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 $size$ of live objects in the old generation: $limit' = f \cdot size$. The next major garbage collection is scheduled when the bytes allocated since the last major garbage collection exceed $limit' - size$.

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

Figure 4 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 $limit'$ based on the heap size $size$. The subsequent garbage collections at time t_2 and t_3 are triggered by the memory reducer. The dotted line shows what the heap size would be without the memory reducer.

The memory reducer can start major garbage collection before the allocation limit is reached. Since this can increase latency, we developed heuristics that rely not only on the idle time provided by the Blink scheduler between frames, but also on whether the webpage is now inactive. We use the JavaScript allocation rate and the rate of JavaScript calls from the browser as signals for whether the webpage is active or not. When both rates drop below a predefined threshold, we consider the webpage to be inactive.

Figure 5 shows the states and transitions of the memory reducer. In the $done$ state the controller waits for a signal that there have been enough allocations performed to warrant a garbage collection. We use a non-idle major garbage collection, triggered by the allocation limit as the signal to transition to the $wait_i$ state. In the $wait_i$ state the controller waits for the webpage to become inactive. When that happens, the controller starts incremental marking and transitions to the run_i state, until the major garbage collection finishes. If heuristics indicate that an additional major garbage collection is likely to reduce memory usage, then the controller transitions to the $wait_{i+1}$ state. Otherwise, it moves back to the initial $done$ state.

To estimate if an additional garbage collection would free more memory we consider:

- The difference between the committed and the used memory after the garbage collection. If the difference is large enough then another garbage collection can compact the heap and release unused memory. The subsequent garbage collection scheduled by the memory reducer will explicitly perform more memory compaction to decrease fragmentation.
- Whether any weak references from Blink to V8 were cleared during garbage collection or not. A cleared weak reference can free objects in Blink, which in turn can remove strong references to the V8 heap. Multiple garbage collections may be necessary to clean up the whole transitive closure of Blink and V8, whose objects are managed by separate heaps.

A well-tuned threshold for the allocation rate is the central part of the controller. Initially we used a fixed threshold, which worked well for the powerful desktop and laptop computers on which we originally tuned the system, but did not work for slower mobile devices. In order to adapt to different hardware speeds, we now consider the allocation rate relative to the measured major garbage collection speed: $\mu = g/(g+a)$, where g is the major garbage collection speed and a is the allocation rate. This ratio μ can be thought of as the mutator utilization for the time window from now until the next major garbage collection, under the assumption that the current allocation rate stays constant and the heap is currently empty:

$$\begin{aligned}
T_{mu} &= \mathit{limit}/a && \text{(mutator time)} \\
T_{gc} &= \mathit{limit}/g && \text{(GC time)} \\
\mu &= T_{mu}/(T_{mu} + T_{gc}) && \text{(mutator utilization)} \\
&= (\mathit{limit}/a)/(\mathit{limit}/a + \mathit{limit}/g) \\
&= g/(g + a)
\end{aligned}$$

This gives us the condition for inactive allocation rate: $\mu \geq \mu_{inactive}$, where $\mu_{inactive}$ is a fixed constant (0.993 in our implementation).

5.4 Predecessors

The `IdleNotification(bool high_priority)` API was already introduced in V8 in Chrome 4 to reduce memory consumption. Chrome called this API after events that may have produced a lot of garbage memory with a given priority to immediatelly free up that memory, e.g., shortly after navigating from one page to another which introduced user observable garbage collection jank for certain page navigation events.

The first version of idle time garbage collection scheduling shipped in Chrome 38, before the Blink task scheduler was implemented. Back then, the compositor, which has a notion of the time taken to prepare frames, invoked V8 whenever it was done with processing a single frame, passing the remaining frame time to an `IdleNotification(int idle_time_in_ms)` API. V8 had to compute if garbage collection was necessary, if it would fit into the given idle time slot, and schedule garbage collection operations accordingly.

This approach showed promising results but had a few limitations. First, it did not allow idle times longer than 16.6 ms, since it was limited by the maximum frame time. Second, it added additional overhead to frame rendering. At the end of each rendered frame, V8 was always invoked regardless of whether there was pending garbage collection work. Third, the compositor had no global notion of idleness in the system. It knows only the time left after rendering a frame, but not whether other tasks have higher priority, e.g. pending user input events. Missing such high priority events may result in dropped frames if garbage collection is performed erroneously during misclassified idle time.

When the Blink task scheduler shipped in Chrome 41, we moved the compositor-driven idle notification implementation over to the Blink scheduler. Whenever the Blink task scheduler detected idleness in the system, it scheduled a V8 idle task which called the `IdleNotification` API as soon as it was scheduled. The Blink scheduler enabled scheduling of idle tasks with deadlines longer than 16.6 ms, due to its global knowledge of whether frames are expected, and what other tasks are scheduled to be run. Moreover it has a global notion of task queue priority preventing lower priority idle tasks from interfering with higher priority latency-critical tasks. However, the system still used the push-based approach of proactively calling the V8 `IdleNotification` API, meaning it still unnecessarily wasted CPU time in cases where V8 has no garbage collection work pending.

The work presented in this paper shipped in Chrome 45. In this version we moved to the pull model described in Section 5, where V8 posts garbage collection idle tasks only when necessary, reducing wasted CPU time. In addition, this enabled implementation of the memory reducer algorithm presented above.

6. Related Work

In this section we are relating our work to (1) systems that take advantage of idle time to provide real-time guarantees and (2) concurrent, parallel, and incremental garbage collectors that reduce garbage collection work on the application threads. A more extensive overview of different types of garbage collectors can be found in [23].

6.1 Idle Time Garbage Collection

Henriksson [21] demonstrated a system where a garbage collection task is scheduled as a special low priority task that runs during idle time. In that system, real-time threads run with the highest priority and are therefore never preempted. However the garbage collection task has to be preemptable to ensure schedulability. Making garbage collection operations preemptable may complicate the garbage collection implementation and may degrade peak performance, due to more expensive read or write barriers to support this feature. Our approach allows scheduling of specific garbage collection operations which are selected based on online profiling of garbage collection operations and application behavior. V8 is not designed for hard real-time requirements, but aims at providing high throughput as a high priority, while taking advantage of idle time to avoid interruptions of the garbage collector and reduce memory consumption when the application becomes inactive. V8 does not support abortion of garbage collection operations, but instead schedules and scales them carefully.

The work of Henriksson was further refined in [30], where garbage collection tasks are given a deadline and an earliest deadline first scheduler [11] makes sure that none of the tasks miss their deadline. However, finding the right deadline for a garbage collection task may be challenging, since, for many garbage collection operations, the workload is unknown before starting (e.g. determining the transitive closure of live objects).

Opportunistic garbage collection [35, 36] schedules garbage collections explicitly after long computationally intensive operations and if the user does not interact with the program for a long time, i.e. several minutes are mentioned in the paper. We think that adding a garbage collection pause to a long computation pause makes user experience even worse. When frames are missed in Chrome because of a long computationally intensive JavaScript operation no idle time is reported to V8 until frame deadlines are again met. Scheduling garbage collections after long inactive periods to improve user experience in upcoming user interactions is related to our memory reducing garbage collections, which we use to reduce memory footprint.

In periodic garbage collection scheduling, garbage collection is performed on a predefined periodic fixed schedule for a predefined amount of time. The application threads are stopped and the garbage collector thread is given the highest real-time priority. Metronome [3, 4] used this approach to provide hard real-time guarantees in a Java virtual machine. Finding the right period and length is difficult but key to provide hard real-time guarantees. TuningFork [5], a tool designed for Metronome users, may help to find such timing parameters for an application based on offline profiling. Depending on the application and its allocation rate, program throughput may degrade significantly while providing schedulability.

In hybrid garbage collection scheduling, garbage collection tasks are scheduled periodically and additionally takes advantage of idle time when needed. Such a system was first implemented in Metronome-TS [2] and showed significant improvement over the regular Metronome system.

Kalibera et al. [24] discuss different approaches to garbage collection scheduling on uniprocessor systems and classifies them in three categories: slack-based [21], periodic [3, 4], and hybrid [2]. The authors compare performance and schedulability tests of different implementations of these systems, where the garbage collector is run on a separate thread to take advantage of the operating system scheduler. They found that on average hybrid garbage collection scheduling provides the best performance, but there exist applications that favor a slack-based or periodic system. Moreover, they found that read and write barriers used in these systems introduce a significant performance overhead and may degrade system throughput significantly.

6.2 Concurrent, Parallel, and 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. Before we discuss different implementations of these techniques let us first introduce their meaning. Concurrent garbage collection means that garbage collection work is performed concurrently to the application threads on separate threads. Parallel garbage collection means that multiple garbage collection threads split up the garbage collection work among them. These threads can additionally run concurrent to the application, or pause the application threads which then have to wait for the garbage collection threads to finish. Incremental garbage collection means that the application threads get interrupted periodically to perform small garbage collection work items.

Real-time guarantees may be achieved with concurrent or incremental garbage collection techniques, which typically require read [6] or write [12] barriers to ensure a consistent heap state. The implementation of such garbage collectors may degrade application throughput due to expensive barrier overhead, increase memory consumption while copying concurrently [12], and increase code complexity of the virtual machine.

The speed-up of parallel garbage collection is limited by the number of CPUs available to the garbage collection threads. The main challenge is to partition the work efficiently using work stealing [16, 31], processor-centric [12] or memory-centric [22] schemes. In [7], another parallel garbage collector is introduced where only a small fraction of the heap is processed at a given point in time. Large cyclic data structures that span multiple heap fractions may cause many extensive reference updates in such a garbage collector which degrades performance.

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, as discussed in Section 2.3. Both operations are also performed during idle time to ensure fast progress of these phases. Most importantly, costly memory compaction phases like young generation evacuation or old generation compaction can be efficiently hidden during idle times without introducing memory or barrier overheads. For a best effort system, where hard real-time deadlines do not have to be met, idle time garbage collection scheduling may be a simple, high throughput, and low memory footprint alternative.

7. Experiments

We ran our experiments with Chrome version 48.0.2564.109 (February 2016) on a Linux workstation and an Android mobile device. The Linux workstation contains two Intel Xeon E5-2680 V2 deca-core 2.80 GHz CPUs and 64GB of main memory. We used a Nexus 6P Android smartphone for mobile experiments, which has 3GB of main memory and a BIG.little configuration of a Quad-core 1.55 GHz Cortex-A53 and a Quad-core 2.0 GHz Cortex-A57. In order to evaluate the impact of running garbage collection during idle

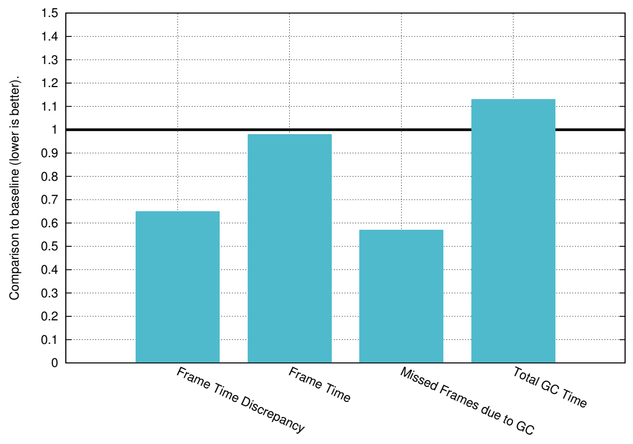


Figure 6: Frame time discrepancy, frame time, number of frames missed due to garbage collection, and total garbage collection time compared to the baseline on the oortonline.gl benchmark.

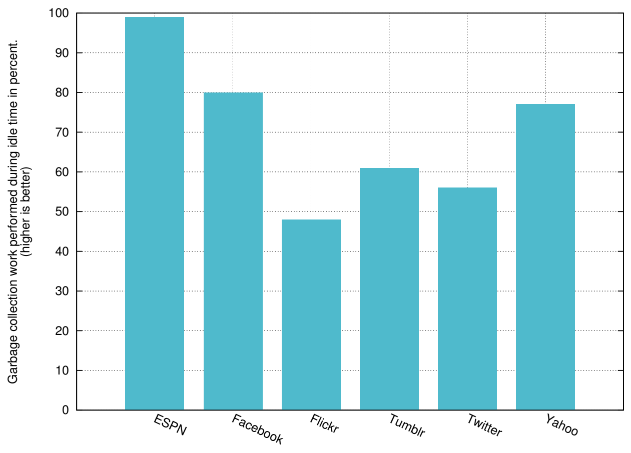


Figure 7: Percentage of garbage collection work performed during idle time.

times, we run popular latency critical and memory critical webpages. We used Chrome's Telemetry performance benchmarking framework to evaluate recorded samples of real webpages. Telemetry provides various jank, memory, and garbage collection related performance counters. Each benchmark was run 20 times for each configuration to provide stable measurement results with low variance.

Idle time garbage collection scheduling is enabled by default in Chrome. As a baseline we use Chrome started with the command line flag `--disable-v8-idle-tasks` to turn off the idle task mechanism. When describing the percentage of idle garbage collection work, a result of 100% represents that all garbage collection work done was done during idle time.

Moreover we present results for frame rate, frame time discrepancy, and total garbage collection time. To quantify the impact of garbage collection on jank precisely we report the number of frames missed due to garbage collection, i.e., all frames which would not have missed their deadline if garbage collection would not have happened.

The PLDI'16 artifact evaluation committee declared the experiments presented below reproducible. A detailed description of how to run the experiments and obtain the presented results can be found in the artifact [15].

7.1 OortOnline.gl

OortOnline.gl is an online WebGL [25] benchmark using the Turbulenz [34] gaming engine that measures the rendering performance of a web browser. The benchmark is started automatically after opening *http://oortonline.gl* and clicking on the start button. After that, four different scenes with animations are rendered. OortOnline.gl is GPU intensive benchmark which did not run correctly on the Nexus 6P due to a GPU shader driver bug, therefore we ran all experiments on a Linux Workstation. We ran OortOnline.gl in Telemetry, where the performance measurement is started right after loading the benchmark and performing an explicitly triggered garbage collection to create a similar heap state for both the baseline and idle time garbage collection scheduling Chrome instances. The measurement is stopped after 30 seconds, shortly before the benchmark ends to avoid shutdown artifacts.

The improvements from using idle time garbage collection scheduling are shown in Figure 6. Frame time discrepancy reduces on average from 212 ms to 138 ms, which corresponds to 0.65 of the baseline. The average frame time of the idle time version is 0.98 of the baseline, which translates to an improvement from 17.92 ms to 17.6 ms. This is a significant improvement, since most of the rendered frames do not contain garbage collection events. The baseline version misses on average 59.8 frames because of garbage collection work, while the idle time version misses 35.1 frames. These results correlate with an average of 85% of garbage collection work being scheduled during idle time, which significantly reduced the amount of garbage collection work performed during time critical phases. However, on average 11% of the garbage collection work scheduled during idle time overshot its deadline. In these cases, our heuristics did not correctly predict the garbage collection workload. In total, idle time garbage collection scheduling increased the total garbage collection time by 13% to 780 ms on average, due to scheduling garbage collection proactively and making faster incremental marking progress with idle tasks. Hence major garbage collections finish earlier which makes it more likely that another major garbage collection will be triggered during the execution time of the benchmark. Memory consumption reduced to 0.91 of the baseline at the end of the benchmark, which corresponds to an improvement from 374M to 340M.

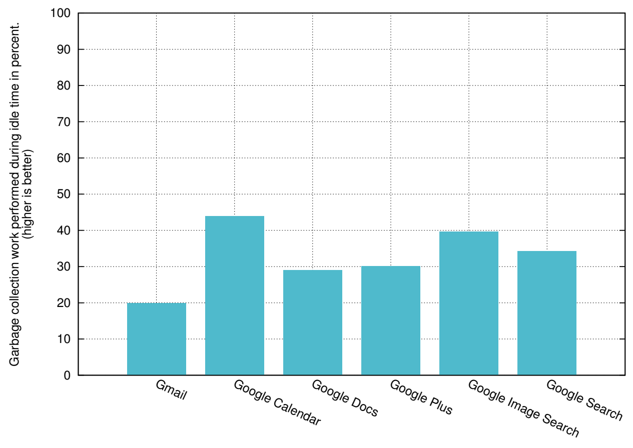


Figure 8: Percentage of garbage collection work performed during idle time.

7.2 Infinite Scrolling

Infinite scrolling is a technique that gives the user the illusion that a webpage has infinite content. New content is created on the fly while the user scrolls down on a webpage. Scrolling such a webpage for several seconds typically creates a continuous stream of newly allocated objects which may trigger several garbage collections. Such garbage collection pauses, triggered while scrolling, may result in visible jank for the user.

We run six popular webpages in the infinite scrolling benchmark: ESPN, Facebook, Flickr, Tumblr, Twitter, and Yahoo. We created a replayable archive of these webpages using Telemetry and instrumented it to trigger a manual garbage collection after page load to create a similar heap state for both the baseline and idle time garbage collection scheduling Chrome instances. The measurement is started after the manual garbage collection, while scrolling the webpages for up to 60 seconds. The scrolling speed is about 700 pixels per second.

Figure 7 shows the amount of garbage collection work performed during idle time. The most garbage collection work during idle time is performed running the ESPN webpage. 99% of the garbage collection work is performed while Chrome is idle. Garbage collection idle time scheduling has the smallest effect on Flicker, which just allows 47% of its garbage collection work being scheduled during idle time. On average, about 70% of the total garbage collection of this set of webpages is performed during idle time. 5% of the scheduled garbage collection operations overshot the given idle deadline. The baseline version misses on average 22.8 frames due to garbage collection work, while the idle time version misses 12.7 frames, which translates to 0.55 of the baseline. This however does not have impact on the frame rate and the frame time discrepancy, with the average frame times of 17.62 ms and the average frame time discrepancy of 291 ms for both configurations. There are various reasons why idle time garbage collection scheduling does not have impact on these metrics. First, the number of frames missed due to garbage collection is only 2.4% of all missed frames. Some webpages like Twitter execute long running JavaScript code on the main thread to build up the page incrementally, which results in high frame time discrepancy regardless of V8's garbage collection events. Second, threaded-scrolling, a technology used in Chrome where scrolling is performed on a separate thread, compensates for some garbage collection pauses on the main thread. We investigated turning off threaded-scrolling, however this configuration was unrealistic and not representative of real-world use-cases. It resulted in average frame times that were much longer than the default configuration due to scrolling blocked by drawing on the main thread.

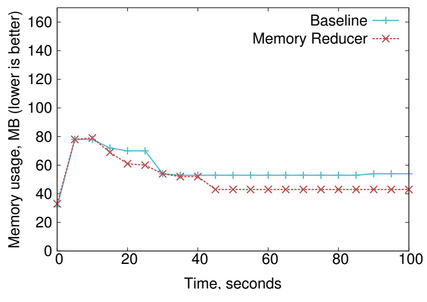
The total garbage collection time on the whole page set is 1.23 of the baseline, due to scheduling garbage collection proactively and finishing major garbage collection earlier, resulting in more major garbage collections. However, performing more major garbage collections reduces memory consumption by 0.91 of the baseline. Most memory was saved on Facebook where memory consumption decreased to 57M on average, which corresponds to 0.81 of the baseline.

In order to see how idle time garbage collection scheduling performs on mobile devices, we run the mobile versions of the webpages on Android Nexus 6P device. Overall about 39% of the total garbage collection work is performed during idle time and 4% of the scheduled garbage collection operations overshoot the given idle deadline. There is less idle time compared to the desktop because the device is less powerful. The percentage of idle time garbage collection is highest on ESPN and Twitter (88%) and is smallest on Tumblr webpage (0.5%). There is no impact on frame times and frames missed due to garbage collection.

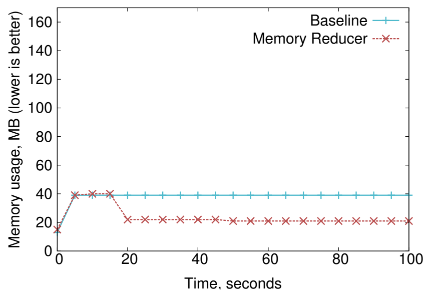
7.3 Page Load with Scrolling

In the page load with scrolling benchmark we measure the execution from the beginning of a page load until scrolling down to the end of the page, i.e., up to five seconds. We run six popular webpages: Gmail, Google Calendar, Google Docs, Google Plus, Google Image Search, and Google Search which were recorded using Telemetry. Note, that none of these webpages implements infinite scrolling.

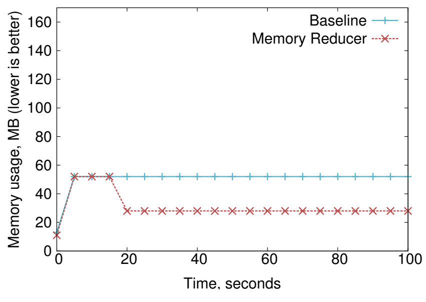
Figure 8 shows the result of this experiment on a Linux Workstation. During page load time, all of the webpages have no idle time. High throughput is important during page load to display the webpage as fast as possible for the user. After that, while scrolling, latency becomes higher priority and idle times may become available. In Google Calendar most of the garbage collection work is performed during idle time, on average 44%. In Gmail on average just 20% of the garbage collection work is performed during idle time. Gmail performs many computations while scrolling, and therefore does not have much idle time to spare at the



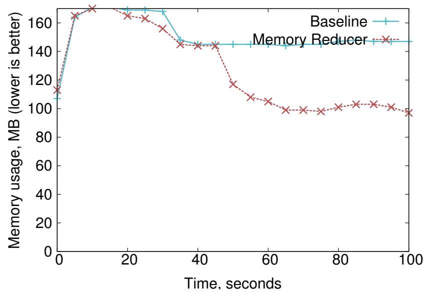
(a) ESPN



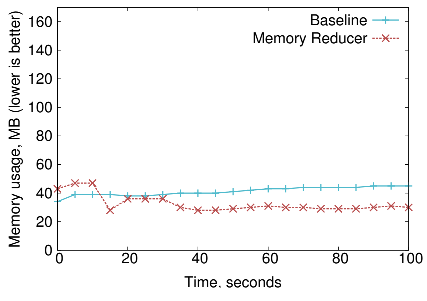
(b) Facebook



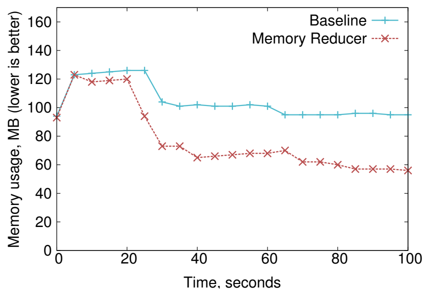
(c) Flickr



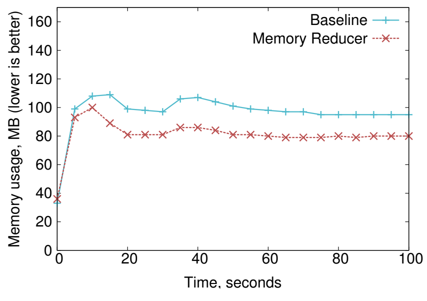
(d) GMail



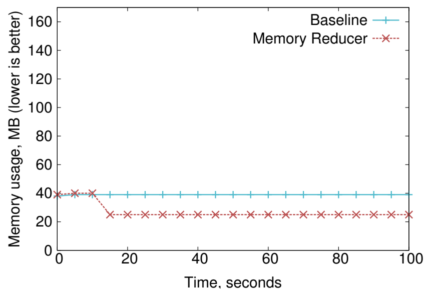
(e) Google Calendar



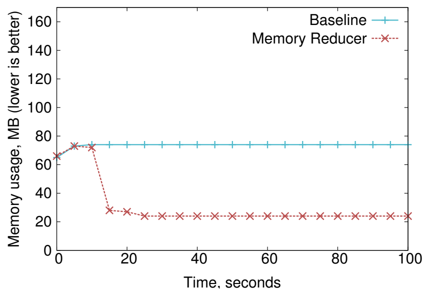
(f) Google Docs



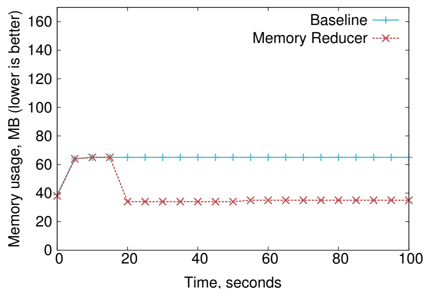
(g) Google Plus



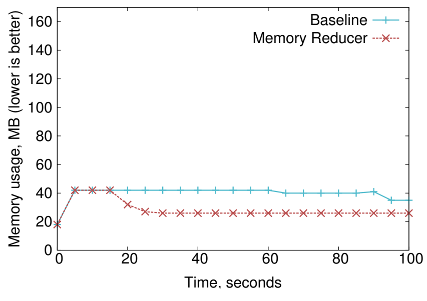
(h) Google Image Search



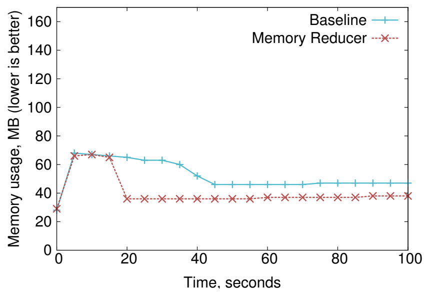
(i) Google Web Search



(j) Tumblr



(k) Twitter



(l) Yahoo

Figure 9: Memory reducer memory usage in comparison to the baseline running idle webpages.

end of each frame. On average 8% of garbage collection operations overshot the idle time limit. The total garbage collection time is 1.24% of the baseline. Again, proactively scheduling garbage collection and finishing major garbage collection earlier is the root cause of the increase. Memory consumption over the whole page set is 0.75 of the baseline.

Note that, in comparison to the infinite scrolling benchmarks discussed in Section 7.2 the ratio of garbage collection during idle time is significantly smaller, because many garbage collection operations happen during page load where no idle time exists. The number of frames missed due to garbage collection is 4 for both configurations. Idle time garbage collection has also no measurable impact on frame times (23 ms), and on frame time discrepancy (1048 ms). This suggests that the frame times are dominated by non-garbage collection pauses.

On Android Nexus 6P only 10% of the total garbage collection work is scheduled during idle time over all webpages. About 2% of the scheduled garbage collection operations overshoot the given idle deadline. The percentage of idle time garbage collection is highest on Google Image Search and GMail (14%) and is smallest on Google Plus webpage (4%). There is no impact on frame times and frames missed due to garbage collection.

7.4 Memory

In the memory benchmark we measure memory consumption of V8 on idle webpages. We load all the webpages used in the previous two benchmarks on a Linux Workstation and leave them open for 100 seconds without any user interaction. As a baseline we run V8 with the memory reducer component disabled (`-js-flags=--nomemory-reducer`), but keep idle task and enabled. Thus the baseline never starts an idle major garbage collection, but it still uses idle tasks to perform minor garbage collections and incremental marking steps for non-idle major garbage collections.

Figure 9 shows how V8 memory usages changes over time for each webpage. The memory usage is sampled every five seconds. In the first few seconds both versions use the same amount of memory as the webpages load and allocate with high rate. After a while the webpages become idle since there no user interaction. Once the memory reducer detects that the webpage is idle, it starts a major garbage collection. At that point the graphs for the baseline and the memory reducer diverge. Note that without the memory reducer, we observed that for many webpages a major garbage collection was triggered only after several hours. Over all webpages the final average memory usage is 0.64 of the baseline. The impact is highest on Google Web Search with the memory usage 0.34 of the baseline and is lowest on Google Plus with the memory usage 0.83 of the baseline.

7.5 Throughput

We measure the impact of idle time garbage collection scheduling on throughput by running the popular JavaScript benchmarks Octane [19], Kraken [29], Speedometer [10], and Jetstream [9]. These benchmarks, except the SplayLatency and MandreelLatency benchmark of Octane, care about fast execution. Long pauses introduced by the virtual machine are not relevant. The benchmark scores on all four benchmarks remain unchanged with and without idle garbage collection scheduling. We see that as a positive result, since we designed idle time garbage collection scheduling to reduce latency and memory consumption, while maintaining high throughput. Idle time garbage collection scheduling does not improve SplayLatency and MandreelLatency because there is no idle time when executing these benchmarks.

8. Conclusions

In this paper we introduced idle time garbage collection scheduling, a garbage collection optimization which increases the responsiveness of applications and reduces memory consumption when a website is inactive by making use of known idle periods to hide garbage collection work in otherwise idle application time. We introduced a new metric, called frame time discrepancy to better quantify user experience in animations and demonstrated performance improvements on that metric in the WebGL based game benchmark oortonline.gl. We showed that for many popular webpages garbage collection operations can be successfully hidden during idle time, without impacting peak performance. We demonstrated that garbage collection scheduled during idle times can significantly reduce memory consumption on inactive webpages. Idle time garbage collection scheduling is implemented and shipped in V8 and Chrome since version 38.

We believe that idle time garbage collection scheduling should also work for other virtual machines using garbage collection in a setting where the virtual machine is aware of screen rendering, e.g. the ART virtual machine [17]. Similarly, idle time garbage collection scheduling could be beneficial on virtual machines which are not constantly under 100% CPU load. Concretely, we think that server applications like node.js [32] that are built on top of V8 could take advantage of the idle garbage collection API and schedule garbage collections when they are idle, such as while waiting on network requests.

References

[1] M. Aigner, T. Hütter, C. M. Kirsch, A. Miller, H. Payer, and M. Preishuber. ACDC-JS: Explorative Benchmarking of Javascript Memory Management. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 67–78, New York, NY, USA, 2014. ACM.

[2] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic Scheduling for Real-time Garbage Collection. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 245–254. ACM, 2008.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298. ACM, 2003.

[4] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling Fragmentation and Space Consumption in the Metronome, a Real-time Garbage Collector for Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 81–92. ACM, 2003.

[5] D. F. Bacon, P. Cheng, and D. Grove. TuningFork: A Platform for Visualization and Analysis of Complex Real-time Systems. In *Companion to the 22nd ACM SIGPLAN Confer-*

ence on Object-oriented Programming Systems and Applications Companion, OOPSLA '07, pages 854–855. ACM, 2007.

[6] H. G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Commun. ACM*, 21(4):280–294, Apr. 1978.

[7] O. Ben-Yitzhak, I. Goft, E. K. Kolodner, K. Kuiper, and V. Leikehman. An Algorithm for Parallel Incremental Compaction. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 100–105. ACM, 2002.

[8] J. Bentley. Programming Pearls: Algorithm Design Techniques. *Commun. ACM*, 27(9):865–873, Sept. 1984.

[9] Browserbench. Jetstream. `http://browserbench.org/JetStream`, . Accessed: 2016-03-15.

[10] Browserbench. Speedometer. `http://browserbench.org/Speedometer`, . Accessed: 2016-03-15.

[11] G. Buttazzo. Red: Robust earliest deadline scheduling. In *Proceedings of the 3rd International Workshop on Responsive Computing Systems*, pages 100–111, 1993.

[12] P. Cheng and G. E. Blelloch. A Parallel, Real-time Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 125–136. ACM, 2001.

[13] D. Clifford, H. Payer, M. Starzinger, and B. L. Titzer. Allocation Folding Based on Dominance. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 15–24. ACM, 2014.

[14] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 105–117. ACM, 2015.

[15] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer. PLDI'16 Artifact: Idle Time Garbage Collection Scheduling. `https://goo.gl/AxvigS`. Accessed: 2016-04-10.

[16] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel Garbage Collection for Shared Memory Multiprocessors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. USENIX, 2001.

[17] Google Inc. Android Runtime (ART). `http://source.android.com/devices/tech/dalvik/index.html`, . Accessed: 2016-03-15.

[18] Google Inc. V8 Design. `https://code.google.com/p/v8/design`, . Accessed: 2016-03-15.

[19] Google Inc. Octane. `https://developers.google.com/octane`, . Accessed: 2016-03-15.

[20] Google Inc. The RAIL performance model. `http://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail`, . Accessed: 2016-03-15.

[21] Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[22] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *Transactions on Parallel and Distributed Systems*, 4(9):1030–1040, 1993.

[23] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Jan. 2012.

[24] T. Kalibera, F. Pizlo, A. L. Hosking, and J. Vitek. Scheduling Real-time Garbage Collection on Uniprocessors. *ACM Trans. Comput. Syst.*, 29(3):8:1–8:29, Aug. 2011.

[25] Khronos Group. WebGL. `http://www.khronos.org/webgl`. Accessed: 2016-03-15.

[26] S. Kyostila and R. McIlroy. Scheduling Tasks Intelligently for Optimized Performance. `http://blog.chromium.org/2015/04/scheduling-tasks-intelligently-for_30.html`. Accessed: 2016-03-15.

[27] R. McIlroy. Cooperative scheduling of background tasks. `http://w3c.github.io/requestidlecallback`. Accessed: 2016-03-15.

[28] R. Miller. Response time in man-computer conversational transactions. In *Proceedings of the Fall Joint Computer Conference*, 1968.

[29] Mozilla Foundation. Kraken Benchmark. `http://krakenbenchmark.mozilla.org`. Accessed: 2016-03-15.

[30] S. G. Robertz and R. Henriksson. Time-triggered Garbage Collection: Robust and Adaptive Real-time GC Scheduling for Embedded Systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 93–102. ACM, 2003.

[31] F. Siebert. Concurrent, Parallel, Real-time Garbage-collection. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 11–20. ACM, 2010.

[32] The Node.js Developers. Node.js. `http://nodejs.org/`. Accessed: 2016-03-15.

[33] S. Tilkov and S. Vinoski. Node.Js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, Nov. 2010.

[34] Turbulenz Limited. Turbulenz Engine. `http://biz.turbulenz.com`. Accessed: 2016-03-15.

[35] P. R. Wilson. Opportunistic Garbage Collection. *SIGPLAN Not.*, 23(12):98–102, Dec. 1988.

[36] P. R. Wilson and T. G. Moher. Design of the Opportunistic Garbage Collector. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 23–35. ACM, 1989.