ACDC-JS: Explorative Benchmarking of JavaScript Memory Management

Martin Aigner[+] Thomas Hütter[+] Christoph M. Kirsch[+]
Alexander Miller[+] Hannes Payer[*] Mario Preishuber[+]

[+]University of Salzburg
firstname.lastname@cs.uni-salzburg.at

[*]Google, Inc.
hpayer@google.com

Abstract

We present ACDC-JS, an open-source[1] JavaScript memory management benchmarking tool. ACDC-JS incorporates a heap model based on real web applications and may be configured to expose virtually any relevant performance characteristics of JavaScript memory management systems. ACDC-JS is based on ACDC [11], a benchmarking tool for C/C++ that models periodic allocation and deallocation behavior (AC) as well as persistent memory (DC). We identify important characteristics of JavaScript mutator behavior and propose a configurable heap model based on typical distributions of these characteristics as foundation for ACDC-JS. We describe heap analyses of 13 real web applications extending existing work on JavaScript behavior analysis [13]. Our experimental results show that ACDC-JS enables performance benchmarking and debugging of state-of-the-art JavaScript virtual machines such as V8 and SpiderMonkey by exposing key aspects of their memory management performance.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Memory Management

General Terms Performance, Measurement

Keywords benchmarking; automatic heap management; program behavior

1. Introduction

JavaScript performance is an important issue for supporting the next generation of web applications. Browser vendors compare their products mainly by presenting JavaScript benchmarking results. As a consequence, development of JavaScript virtual machines is driven by industry-standard benchmarking suites. However, recent studies suggest that such benchmarks do not reflect the behavior of real web applications [13]. Speedups based on such benchmarks do therefore not necessarily translate into speedups of real web applications [16] since JavaScript implementations are tuned to perform well on such benchmarks. Furthermore, the benchmark suites only cover a small set of possible workloads.

One important issue in JavaScript performance is memory management which affects all important performance dimensions namely throughput and latency as well as memory consumption. However, the most common JavaScript benchmarking suites Octane (Google) [5], Kraken (Mozilla) [4], and SunSpider (WebKit) [9] focus mainly on throughput and only two recent benchmarks in Octane (SplayLatency and MandreelLatency) provide information on garbage collection latency and compiler latency, respectively. No memory consumption is reported by any benchmarking suite because there is no portable way to measure memory consumption from within JavaScript running in a browser.

We propose ACDC-JS, an open source JavaScript memory management benchmarking tool based on ACDC which is a tool for benchmarking memory allocators for C/C++ [11]. ACDC-JS implements an configurable allocation model that allocates objects of different sizes periodically (AC) as well as permanently (DC). Additionally, ACDC-JS may be configured to create heap structures to emulate data structures found in real applications. ACDC-JS reports throughput and latency of memory allocation and access as well as memory consumption. ACDC-JS may thus expose tradeoffs of implementation details of JavaScript garbage collectors.

In this work we extend recent studies on the allocation behavior of real JavaScript applications [13] and use this information to implement a configurable mutator model for benchmarking JavaScript memory management systems. Although we believe that it is not possible to have *the* realistic benchmark, we do believe that having a configurable workload generator based on properties found in real web applications is useful to cover a large set of possible and meaningful workloads. We also verify experimentally that the allocation behavior of ACDC-JS is comparable to the average allocation behavior of the real JavaScript applications.

Our experimental evaluation of the state-of-the-art JavaScript virtual machines V8 and SpiderMonkey shows that ACDC-JS is capable of exposing differences in all relevant performance dimensions as well as exposing trade-offs of different garbage collection policies implemented in different configurations of the VMs. The evaluation follows a 4-step process for performance debugging that begins with selecting an expected performance characteristic of a given memory management feature, e.g., high allocation throughput with generational GCs on workloads with mostly short-living objects. We then design a workload that exposes this characteristic and, ideally, still resembles the memory usage of real applications. The third step is to configure ACDC-JS such that it emulates this workload. The final step is to run ACDC-JS in this configuration and compare the result with our expectations.

[1] https://github.com/chromium/ACDC4GC

DLS '14, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2661088.2661089

In total, we present three experiments: (1) throughput of generational garbage collection, (2) latency of incremental marking, and (3) performance robustness against varying object sizes. The first experiment confirms the performance advantage of generational garbage collection on workloads with mostly short-living objects. This experiment also reveals the impact of incremental marking on throughput, latency, and memory consumption in the presence of long-living objects. The second experiment follows up on the first with an in-depth analysis of the full throughput, latency, and memory consumption trade-off with incremental marking, and in turn reveals the cost of barriers for generational garbage collection. The third experiment checks performance robustness against varying object sizes and reveals a potential performance bug in V8 that produces large variations in latency for neighboring size classes.

This paper makes the following contributions: (1) An analysis of popular and JavaScript-intensive real web applications to obtain distributions of object size and heap structure properties (confirming existing studies on the distribution of object types and lifetimes) and (2) a JavaScript implementation of the ACDC benchmarking tool that is enhanced to incorporate (1) and capable of exposing performance differences and anomalies in various configurations of V8 and SpiderMonkey by exploring a large range of workloads.

2. Related Work

Given the importance of JavaScript for programming web applications, there are surprisingly few papers on JavaScript performance analysis. Developers of JavaScript implementations rely on benchmark suites like Octane (Google) [5], Kraken (Mozilla) [4], and SunSpider (WebKit) [9] to evaluate the performance impact of certain implementation details. However, behavioral analysis and comparison to the behavior of real-world applications suggest that the benchmark suites are not representative of real workloads [13, 14, 16].

Recent literature [16] proposes a record-and-replay approach to create macro benchmarks automatically based on real web applications. While this approach improves the representativeness of a benchmark it does not allow to create workloads for exposing certain performance characteristics of a JavaScript implementation. ACDC-JS, on the other hand, aims at preserving the representativeness of real world applications (macro benchmarks) while also providing the ability to test the performance of specific implementation details (micro benchmarks). However, since ACDC-JS can never emulate all possible real applications we believe that replaying traces of real applications can be a meaningful extension to the synthetic nature of ACDC-JS.

Previous work on real JavaScript application behavior [13, 14] performed an extensive analysis in terms of the execution of functions and code, heap allocation of objects and data, and the performance of event handling. In our work we focus on heap allocation behavior and extend previous work [13] with an analysis of object size distribution and heap graph properties to provide a more detailed allocation model that covers all relevant aspects of memory management performance in JavaScript implementations.

Another approach to extend the understanding of JavaScript behavior focuses on the use of dynamic language features, e.g., changing the prototype chain, adding or deleting properties, or the use of eval, rather than the utilization of dynamic memory [15, 17]. Although ACDC-JS does not aim at modeling dynamic features of JavaScript we rely on the conclusion that they are rarely used in real applications and therefore do not employ such features in our implementation with the additional benefit of avoiding their side effects on ACDC-JS's execution.

The approach of model-based performance analysis is based on ACDC [11], here denoted ACDC-C, where empirical data on properties of real applications allows the implementation of a workload model that preserves representativeness of benchmarks but can also be configured to expose corner cases in the performance of memory management systems. However, ACDC-C focuses on explicit allocation and deallocation in C/C++. Our model requires different parameters and a different implementation to represent memory management workloads of JavaScript applications. Section 4 gives a short overview of ACDC-C and describes our adaptations towards ACDC-JS.

A lot of work has been published on benchmarking Java virtual machines including their memory management performance. The DaCapo benchmark suite [12] seems to be the gold standard for Java performance analysis today. However, the DaCapo suite is a selection of real applications covering a large range of workloads and thus very different from ACDC's approach. We have ported ACDC-C to Java as well but a release incorporating our experiences with ACDC for Java is future work. To the best of our knowledge there is also no benchmarking suite for other managed languages that is similar to our approach.

3. JavaScript Heap Analysis

Modeling a configurable allocation behavior requires understanding the properties of the allocation behavior of real-world applications. Recent work on JavaScript application behavior [13, 14, 16] provides empirical data of real web applications and shows that state-of-the-art benchmarks do not reflect their allocation behavior. Unfortunately, not all aspects that we would like to model in ACDC-JS are covered, namely distribution of object sizes and structural information of typical JavaScript heaps. In this section we describe our analysis of real-world JavaScript applications and present the results that we incorporate in our allocation model of ACDC-JS.

Our heap analysis is based on sampling the JavaScript heap in regular intervals. The applications and user interactions we have analyzed are listed in Table 1. For comparability, the selection is similar to previous work on JavaScript application behavior [13].

We utilize the heap snapshot facility of the Chrome DevTools [3] which is part of the V8 JavaScript virtual machine. Note that we modified V8 to create a heap snapshot automatically at a sample rate of 4 KB, i.e., a snapshot is created whenever 4KB of new objects are allocated by the application. Taking a heap snapshot triggers a full GC cycle so the resulting snapshot is a graph representation of all reachable objects on the JavaScript heap. It contains information about the size of the objects, their type, and also the structure of the heap through references between the objects. The web applications are executed in the Chromium browser running our modified version of V8 and the user interactions are automated by the Selenium browser automation framework [8]. Note that performing our analysis based on V8 might have side effects not visible in other VMs. However, we believe that for our purpose the abstraction of the heap snapshots is portable enough to create a configurable mutator model.

For our model, we are interested in the distribution of object types, sizes and lifetimes, the number of outgoing edges, and the distance from the GC roots to the objects. For each property, we now describe the way in which we have retrieved the distribution from the snapshots and discuss the results.

The object type distribution is a histogram of the types of all objects allocated in each workload. The object type is provided in the heap snapshots. No additional processing of the snapshot data was necessary. Figure 1 illustrates the object type distribution for all workloads. Note that the heap snapshots also contain information specific to the V8 virtual machine and the DevTools heap snapshotting mechanism, namely hidden classes that describe object properties in V8, and so-called synthetic nodes that represent GC roots. We disregard such information here. The type distribution in our

Site	User interaction
CNN `cnn.com`	Read start page news, switch to category *Europe*, read first article of *Top Europe Stories*.
The Economist `economist.com`	Read start page news, switch to category *Science & technology*, read first article.
ESPN `espn.com`	Read start page news, switch to *NASCAR*, click on *Results* and read site.
Hotmail `hotmail.com`	Sign in, check inbox, send email, read an email, delete it, and sign out.
Gmail `www.gmail.com`	Sign in, check inbox, send email, read an email, delete it, and sign out.
Bing Search `bing.com`	Search for *New York* and look at resulting images and news.
Google Search `google.com`	Search for *New York* and look at resulting images and news.
Facebook `facebook.com`	Login and post a message.
Google+ `plus.google.com`	Login and post a message.
Bing Maps `maps.bing.com`	Search for directions from *Austin* to *Houston* by car and walk.
Google Maps `maps.google.com`	Search for directions from *Austin* to *Houston* by car and walk.
amazon `amazon.com`	Search book *Quantitative Computer Architecture*, add to shopping cart, look at cart.
eBay `www.ebay.co.uk`	Search for the book *Quantitative Computer Architecture*.

Table 1: User interactions performed for the heap analysis of real web applications. The selection is based on the work of Livshits et al. [13].

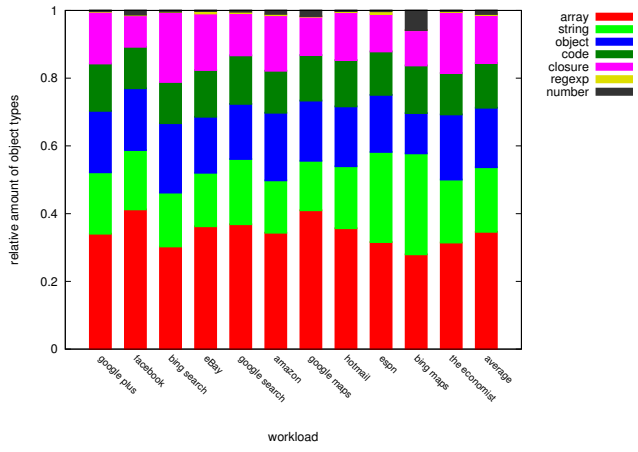

Figure 1: Histogram of the object type distribution for all workloads, confirms the work of Livshits et al. [13].

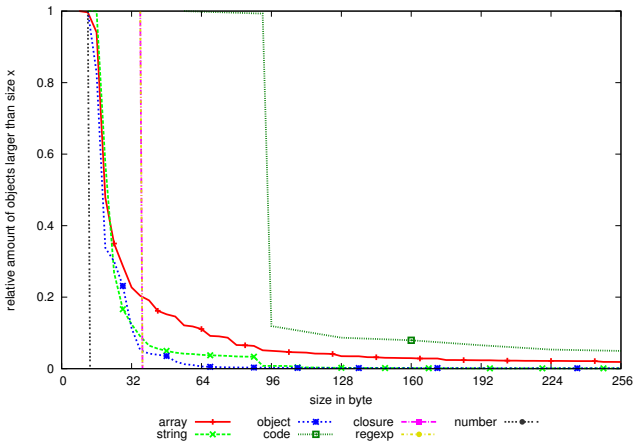

Figure 2: Size distribution of real web applications.

analysis is dominated by arrays, strings, and user-defined objects. The analysis of object type distribution in the work of Livshits et al. [13] appears to record less arrays than we do in our analysis. The discrepancy is caused by growing arrays. Whenever an array grows in a way that requires copying the original content, we treat the new array as a logically new heap object. This, of course, increases the number of allocated arrays and shortens their average lifetime, but we believe that from the point of view of the memory management system a copied or moved array is in fact a new object because the old array can be reclaimed by the GC. This policy is also reflected in the results on the lifetime of arrays below.

Objects in the heap snapshots have a unique identifier. Moreover, the heap snapshots also contain the size of each object. Hence we are able to count the number of objects of a certain size of all snapshots of all workloads. The results are illustrated in Figure 2. On the x axis we have the size in bytes and on the y axis the relative amount of objects that are larger than the size on the x axis. Since the size heavily depends on the object type, we give the size distribution for each type separately. The results are similar to the size distribution in allocation intensive C programs [18] in the sense that small objects occur very frequently whereas the amount of large objects is small. The fixed sizes for numbers, regular expressions (regexp), and closures are V8 specific and will not be considered in our model.

For our analysis of object lifetime, we count the number of subsequent snapshots where an object with a certain identifier occurs. As a consequence, the accuracy of our object lifetime analysis is limited by the snapshot sampling rate of 4 KB. We present the object lifetime distribution in allocated bytes rather than in survived snapshots to be independent from the snapshot sampling rate. By allocated bytes we mean the amount of memory newly allocated after a given object is allocated until this object is identified as dead. The results are illustrated in Figure 3. The x axis gives object lifetime in KB and the y axis gives the relative amount of objects with a lifetime longer than x for each object type. Compared to the work by Livshits et al. [13] arrays live shorter in our analysis for the same reason discussed above: we treat growing arrays as new objects resulting in a higher occurrence of arrays with shorter lifetime.

We are also interested in the structure of a typical JavaScript heap. Therefore we analyze the number of outgoing edges of the heap nodes, called out-degree. Since heap snapshots are graph representations we directly derive this property from the snapshots. Figure 4 shows the out-degree distribution of the analyzed applications for each object type. On the x axis we have the out-degree

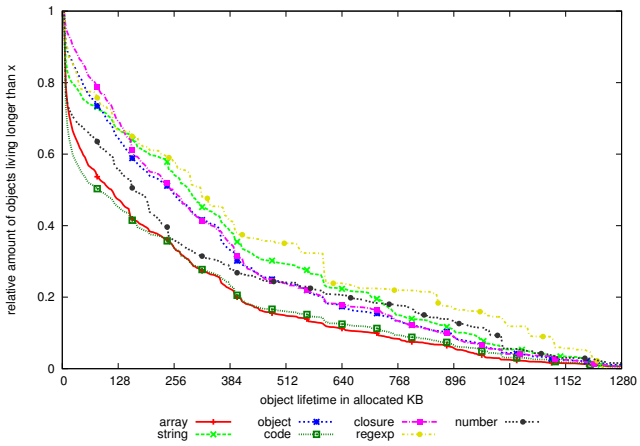

Figure 3: Lifetime distribution of real web applications, confirms the work of Livshits et al. [13].

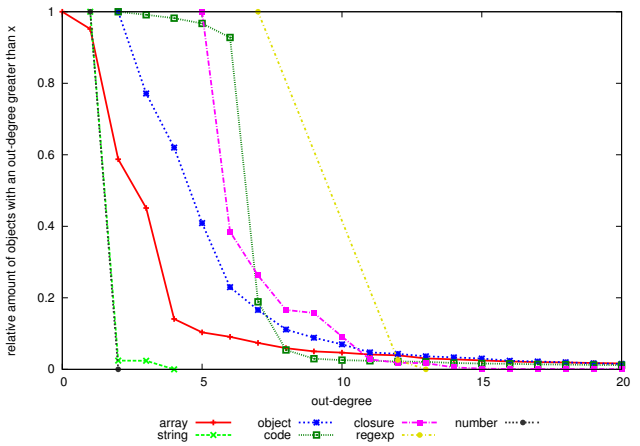

Figure 4: Out-degree distribution of real web applications.

and on the y axis the relative amount of objects with an out-degree greater than x. Most arrays have a low out-degree which suggests that JavaScript arrays usually contain primitive types rather than references to other JavaScript objects.

The in-degree of the graph nodes is also directly derived from the heap snapshots and presented in the same dimensions as the out-degree. The results illustrated in Figure 5 suggest that most objects are referenced by only a few other objects.

Another structural heap property we have analyzed is the shortest distance from the GC roots to an object. The heap snapshots contain special nodes to represent the GC roots. Starting from these nodes we performed a depth first search to all nodes of each workload to obtain the minimal root distance. The results for the root distance distribution are illustrated in Figure 6. On the x axis we have the minimum root distance and on the y axis we present the relative amount of object with a minimum root distance greater than x for each object type. This suggests that 80% of the objects can be traced from the roots in less than 10 steps.

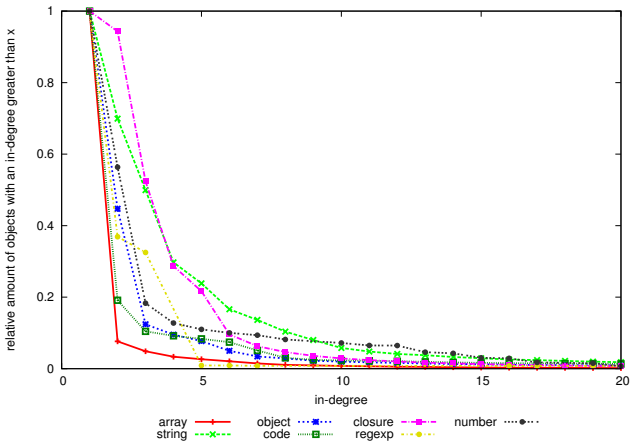

Figure 5: In-degree distribution of real web applications.

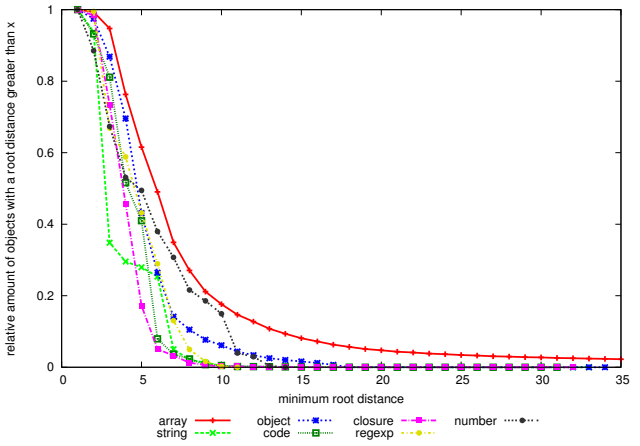

Figure 6: Minimum root distance distribution of real web applications.

4. Overview of ACDC-C and ACDC-JS

Our approach to benchmarking JavaScript implementations is based on ACDC-C [11], a multi-core scalable mutator emulation of allocation intensive C/C++ programs for benchmarking explicit memory management systems. In this section we will give a brief overview of ACDC-C and describe our modifications and extensions that enable a detailed analysis of the memory management in JavaScript virtual machines.

ACDC-C implements an allocation, object sharing, access, and deallocation model based on the behavior of real-world applications. Moreover, ACDC-C allows to configure models for exploring corner cases of mutator behavior. Size and liveness of allocated objects are determined based on random distributions that reflect a behavior where more small and short-living objects than large and long-living objects are allocated. For that, ACDC-C implements a logical notion of time that advances whenever a configurable amount of memory called the time quantum has been allocated.

ACDC-C distinguishes object liveness and lifetime. Object liveness is the time between allocation and last access whereas object lifetime is the time between allocation and deallocation. The time between last access and deallocation is called deallocation delay and emulates deficiencies in identifying dead objects.

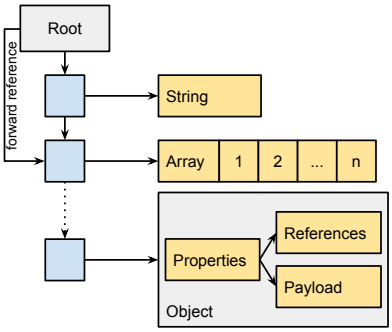

Figure 7: A list-based lifetime-size-class. List nodes support different object types and forward references emulate root distance.

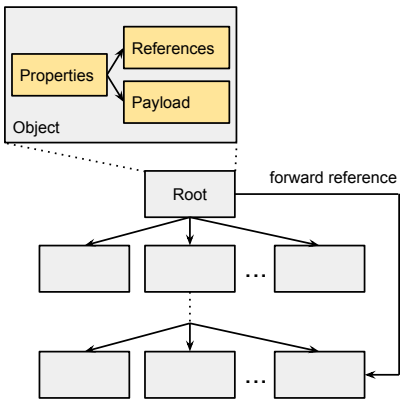

Figure 8: A heap-based lifetime-size-class. Forward references emulate root distance.

ACDC-C collects detailed per-thread information on allocation, access, and deallocation performance. For each allocation and deallocation it counts the number of CPU cycles spent in the malloc and free functions of the allocator. After each newly allocated time quantum it also counts the CPU cycles spent in accessing the live objects. This allows to not only compare the allocation and deallocation performance of allocators and their scalability to multiple threads but also indicates the quality of the memory layout created by the allocators.

The experiments [11] suggest that ACDC-C is indeed capable of exposing differences in all relevant performance dimensions.

While both C/C++ and JavaScript are general purpose programming languages, they differ in many aspects and usually serve different purposes. The remainder of this section discusses their similarities and differences as well as the resulting modifications of ACDC-C that lead to the allocation model of ACDC-JS.

Our analysis of real-world JavaScript programs and related work [13] suggests that the distribution of object size and liveness in JavaScript programs is very similar to C/C++ programs. We therefore reuse the size and liveness model of ACDC-C.

JavaScript has no language support for concurrency and even recent asynchronous programming models like web workers [10] do not support a shared state but implement a message passing paradigm. As a consequence, we disregard all multi-threading and object sharing concepts of ACDC-C.

Memory allocators for C/C++ are not aware of the object type an allocated chunk of memory is supposed to represent in the application. This is not the case in JavaScript implementations where, e.g. number objects might be allocated differently than string objects. ACDC-JS therefore needs to support allocation of different data types. Our analysis of real-world JavaScript applications combined with previous work [13] allows us to model such an allocation behavior.

For ACDC-JS we have also extended the structure of the heap allocated by ACDC-C since our heap structure analysis suggests that the original ACDC-C model is not sufficient anymore. ACDC-C models a very simple relation between allocation and access ordering by gathering objects of the same size and liveness in so-called lifetime-size-classes which are either lists (accesses happen in allocation order) or binary trees (allocation in pre-order, left to right, and accesses in pre-order, right-to-left). ACDC-JS also supports list-based lifetime-size-classes but extends the tree-based lifetime-size-classes to support more than two child nodes to emulate objects containing multiple references, shortcuts from the lifetime-size-class root to certain elements to control the distance between the objects and the GC roots, and back references from certain elements to elements at higher levels to emulate cycles in the object graph. We call such structures heap-based lifetime-size-classes. Similar to tree-based lifetime-size-classes in ACDC-C, the access order of elements in heap-based lifetime-size-classes is different from the allocation order because shortcuts and cycles are randomly added. As a consequence, we preserve the configurability of allocation-order access versus out-of-order access but are also able to emulate complex heap structures that may occur in real-world applications. Figure 7 and 8 illustrate the structure of the lifetime-size-classes implemented in ACDC-JS.

Finally, JavaScript does not offer a portable way to gather performance metrics at the same level of granularity than C/C++. We discuss the limitations in detail in Section 5.1.

5. Implementation Details of ACDC-JS

We reuse the model for typical object size and liveness distributions from ACDC-C [11] since the JavaScript heap analysis suggests that the size and liveness distributions of real applications are similar, in this regard, to those of C/C++ programs. However, we extend the model with the type distribution discussed in Section 3. The types dominating typical JavaScript heaps are arrays, strings, and objects. We adapt the allocation model of ACDC-C accordingly to account for the relative amount of occurrences of these types.

For emulating realistic memory allocation at runtime ACDC-JS first selects an object size from a uniformly distributed, discrete interval $[2^r, 2^{r+1}]$ where r is selected from a uniformly distributed, discrete interval $[log_2(\text{min. size}), log_2(\text{max. size}))$. Next, ACDC-JS selects an object liveness from a uniformly distributed, discrete interval [min. liveness, max. liveness] and adds the configured deallocation delay to obtain an object lifetime. Then, ACDC-JS calculates the number of objects to be allocated based on the previously selected size and liveness with the following formula defined in [11]:

$$\text{number of objects} = (log_2(\text{max. size}) - log_2(\text{selected size}) + 1)^2 * (\text{max. liveness} - \text{selected liveness} + 1)^2$$

Finally, ACDC-JS also emulates type distribution. In particular, ACDC-JS randomly selects an object type to be either a string, an array, or a user-defined object. The average type distribution of real applications illustrated in Figure 1 shows about the same number of occurrences of strings and user-defined objects but about 1.5 times more arrays. ACDC-JS approximates this type distribution by multiplying the number of objects with that factor if the randomly selected type is an array.

Note that we do not account for types when selecting size and liveness to keep the allocation model simple and because the size and lifetime distributions show a similar trend for either arrays,

Parameter	Value
time quantum	1024 KB
benchmark duration	100
min. size	8 B
max. size	64 B
min. liveness	1
max. liveness	10
min. root distance	1
max. root distance	20
min. number of references	0
max. number of references	5
list-based ratio	determined by type distribution
access live objects	TRUE
read-only access	FALSE

Table 2: Default settings

strings, and objects. A more detailed allocation model of ACDC-JS accounting for different liveness based on type information remains future work.

Strings and arrays are implemented as standard JavaScript types. User-defined objects types are implemented with a common object representation illustrated in Figure 7. *Payload* represents the random object size and *References* models the out-degree (discussed below).

ACDC-JS implements lifetime-size-classes different from ACDC-C where the list and tree references are stored in the object's payload. Since JavaScript does not allow pointer arithmetic, list-based lifetime-size-classes require separate list nodes as illustrated in Figure 7.

Heap-based lifetime-size-classes, illustrated in Figure 8, differ from tree-based lifetime-size-classes in ACDC-C to support more than two child objects. The size of *References* is chosen randomly between the parameters min. number of references and max. number of references upon allocation to approximate the out-degree distribution with a common object model. Heap-based lifetime-size-classes also emulate cycles by randomly selecting a cycle length between the parameters min. cycle length and max. cycle length. Heap levels with a distance of the randomly selected cycle length are connected through a back reference to emulate cycles in a JavaScript heap.

Another extension to both types of lifetime-size-classes is emulating the root distance distribution by randomly selecting a root distance between the parameters min. root distance and max. root distance. The root distance distribution is approximated by forward references to nodes having a root distance of a multiple of the randomly selected root distance.

The default parameters of ACDC-JS are listed in Table 2. For all experiments described in Section 6 we use these settings unless stated otherwise.

5.1 Metrics and Probes

Along with total script execution time, ACDC-JS also reports temporal metrics in average allocation time and average memory access time. The data is collected from within ACDC-JS. We keep track of the time it takes to allocate each time quantum and—in case ACDC-JS is configured to access memory—the time required to traverse (and optionally also write) all live lifetime-size-classes after a time quantum has been allocated. For both the allocation time and access time samples, we report the arithmetic mean and the sample standard deviation. The allocation and access time jitter is also reported as the coefficient of variation, i.e., standard deviation divided by arithmetic mean.

In JavaScript there is no way to retrieve information about memory consumption from within the mutator. Therefore we sample the resident set size reported by the operating system at a sample rate of 1 millisecond. We also report the maximum resident set size over an execution of ACDC-JS.

5.2 Accurate Time Measurement in JavaScript

The most portable time measurement facility in JavaScript is the Date.now() method [2] which is supported by all major JavaScript implementations. However, it only provides millisecond resolution. As a consequence, modern browsers support high-resolution time measurement through the performance.now() method [7], specifying microsecond accuracy. Unfortunately, the stand-alone SpiderMonkey shell does not provide the performance object. In order to report time with the highest resolution possible, we employ the PerfMeasurement object [6] available for SpiderMonkey on Linux. PerfMeasurement reports CPU cycles which we scale to microsecond-accurate time values using the clock speed of our experimental environment. We have checked the timing results of a simple JavaScript loop without GC interference to verify that performance.now() in V8 and PerfMeasurement in SpiderMonkey provide comparable results.

6. Experimental Evaluation

All experiments ran on a desktop machine with an Intel i5-2400 4-core 3.1 GHz processor, 32 KB L1 and 256 KB L2 data cache per core, 6 MB shared L3 cache, 8 GB of main memory, and Linux kernel version 3.2.0. We repeated each experiment 10 times. For each metric we report the arithmetic mean and sample standard deviation of the repetitions.

The goal of our experiments is to demonstrate the capabilities of ACDC-JS and not to provide a qualitative comparison of JavaScript virtual machines although the differences between the systems under test might be interesting for VM developers. To expose the performance impact of different implementations and configurations we run ACDC-JS directly on V8 and SpiderMonkey. Both virtual machines can easily be configured to enable or disable certain GC features. We also run ACDC-JS in Chrome and Firefox to study the performance impact of the actual browsers around V8 and SpiderMonkey.

The experimental analysis follows a 4-step process. First, we pick an expected performance characteristic of a particular memory management feature. Second, we define a workload that exposes this very characteristic and still complies with typical JS memory usage found in our heap analysis in Section 3. The third step is to configure ACDC-JS to emulate the workload. The fourth and final step is to measure the performance quantities in actual runs on the JavaScript VMs and compare the results with our expectations. Contradictory results indicate potential performance bugs.

We demonstrate ACDC-JS's capabilities on three performance characteristics: (1) memory management throughput for increasing object liveness, (2) memory management latency for increasing heap size, and (3) memory management robustness for increasing object sizes. The throughput experiment reveals speedup characteristics and promotion thresholds of generational GCs and also illustrates the throughput, latency, and memory consumption trade-off with incremental marking. The latency experiment follows up on that trade-off and provides an in-depth analysis which also reveals in turn the cost of write barriers with generational GCs. The robustness experiment checks whether throughput, latency, and memory consumption are continuous in object size. This experiment reveals a potential performance bug in V8. Finally, we also provide an experiment that does not intend to reveal performance differences in VMs but instead verifies that ACDC-JS's allocation behavior is in-

deed comparable to the object size and lifetime distributions obtained from the 13 real web applications in Section 3.

We compiled and executed the JavaScript virtual machines in the following configurations and labeled the graphs accordingly: V8 can be configured through start-up flags so all V8 results are obtained from the same build using the default compilation settings. V8 represents the default configuration of V8, i.e., a generational GC with incremental marking in the old generation and a semi-space collector in the new generation. V8N runs V8 without incremental marking (flag: –noincremental-marking) and V8E runs V8 in a mode that eagerly compacts the old generation after each GC run (flag –always-compact and –noincremental-marking). Note that V8E is rather a debugging configuration of V8 than a production setting. We anyway include it in our evaluation to increase the set of different GC policies. All V8 configurations are executed in the d8 shell version 3.24.28.

SpiderMonkey requires compile time parametrization. All SpiderMonkey builds share the compile time settings used at "Are We Fast Yet?" [1], a framework for automatically running JavaScript benchmarks on popular virtual machines[2]. SM represents this default configuration, i.e., an incremental mark-sweep collector. SMN runs SpiderMonkey without incremental marking (flag: –disable-gcincremental) and SMG enables a generational GC (flags: –enable-exact-rooting and –enable-gcgenerational). Note that the non-default features of SpiderMonkey might also be still experimental but since we are not interested in a qualitative comparison of JavaScript VMs, we include these settings to demonstrate ACDC-JS's capabilities on various GC policies. Furthermore, SM apparently employs a policy where incremental marking is performed only through the event loop effectively shifting the problem of pause times to where it cannot be triggered by ACDC-JS. All SpiderMonkey configurations are executed in the js shell version JavaScript-C27.0.

For the configurations of Chrome and Firefox we executed ACDC-JS in the browser embedded in a simple HTML page included in the ACDC-JS framework. We expect similar results for Chrome compared to V8 and for Firefox compared to SpiderMonkey. We include the results to also visualize possible performance impacts of embedding the JavaScript virtual machines in the browsers. Note that some results might differ from the bare machine results since we ran the latest stable browser versions which are shipped with different JavaScript engine versions. Another reason for possible differences in performance is that browsers may call the garbage collector explicitly and that collection policies might be different in the shell than when running in a browser. Furthermore, the memory consumption of the browsers contain the whole browser footprint and not only the JavaScript virtual machine. We present results for Chrome version 32.0.1700.17 shipped with V8 version 3.22.24.17 and for Firefox version 27.0 shipped with SpiderMonkey version 24.2.

JavaScript VMs may introduce non-determinism through adaptive techniques like concurrent code optimizations. For all VM configurations we nevertheless do not disable features that might add non-determinism to the execution of ACDC-JS since we are interested in VM behavior in production environments. However, we address non-determinism statistically by replicating experiments 10 times and reporting the sample standard deviations. We have observed that a higher number of replications has only negligible impact on the sample standard deviation. We therefore believe that for our setup 10 replications yield a meaningful estimate of the actual deviation. Furthermore, we have run all experiments with a V8 configuration that disables the optimizing compiler (d8 flag –nocrankshaft) where we observed different absolute performance characteristics but the same relative differences in V8, V8N, and V8E. Additionally, we have run all experiments in V8's predictable mode (d8 flag –predictable) and compared the results to V8, V8N, and V8E running in default mode. Here we did not observe any notable differences in the results in any performance dimension.

[2] SpiderMonkey./configure flags: –enable-optimize –disable-debug –enable-threadsafe –with-system-nspr

Parameter	Value
max. size	8 B
min. liveness	increasing from 1 to 20
max. liveness	min. liveness
time quantum	64 KB
benchmark duration	200
access live objects	FALSE

Table 3: Non-default ACDC-JS settings for the throughput experiment.

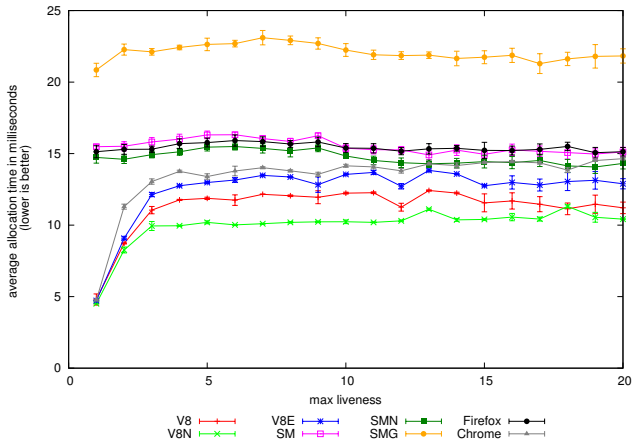

Figure 9: Average allocation time for an increasing object liveness.

6.1 Capabilities of ACDC-JS: Throughput

In this experiment we demonstrate ACDC-JS's capability to provide detailed allocation throughput information for generational garbage collection. Generational garbage collection exploits application predisposition for allocating short-living objects (young generation) at the expense of delaying the reclamation of long-living objects (old generation). We expect higher allocation throughput for short-living objects which is equivalent to lower allocation time since ACDC-JS running on the JavaScript VMs represents a closed system.

We select a workload that gradually increases the liveness of the objects from one time quantum (64 KB) to 20 time quanta (1280 KB). These settings reflect our results from Section 3 where 50% of all objects live shorter than 64 KB and virtually all objects live shorter than 1280 KB. ACDC-JS's benchmark duration is set to 200, i.e., 200 time quanta will be allocated before ACDC-JS terminates. This ensures that even for a liveness of 20 (where ACDC-JS reaches a steady state after 20 time quanta) ACDC-JS executes in a steady state for 90% of the benchmark duration providing data unbiased by a warm-up phase. The object size is fixed to 8 bytes to isolate the impact of object liveness. Table 3 gives an overview of the non-default ACDC-JS settings for this experiment.

The allocation performance for increasing object liveness is presented in Figure 9 where the y axis shows the average allocation

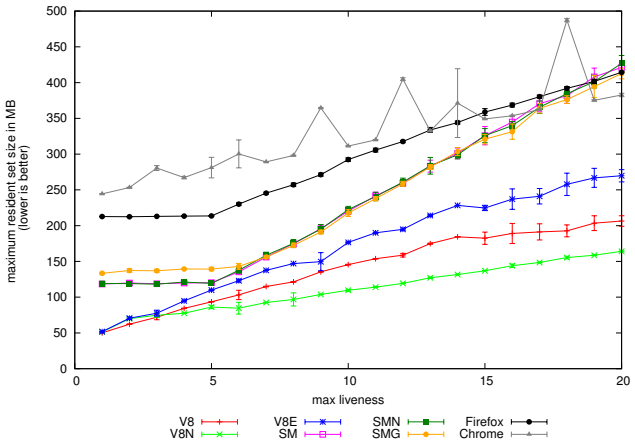

Figure 10: Maximum resident set size for an increasing object liveness.

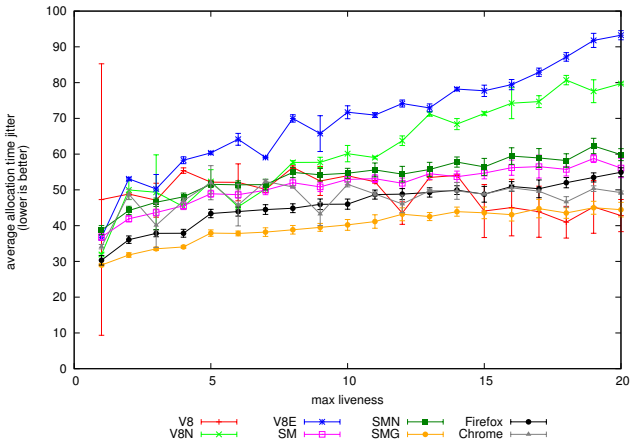

Figure 11: Average allocation time jitter (coefficient of variation) for an increasing object liveness.

time per time quantum in milliseconds. Note that lower allocation time implies higher allocation throughput in our setup. The generational garbage collection implemented in V8, V8N, V8E, and Chrome provides a significant speedup for allocating short-living objects. This suggests that ACDC-JS is indeed capable of creating a workload that exposes the performance impact of generational garbage collection. The SpiderMonkey configurations (SM, SMN, and Firefox) do not enable generational garbage collection and thus show nearly constant allocation performance for all object liveness settings. Although SMG enables generational garbage collection we do not observe better allocation performance for short-living objects. We account this to the fact that at the moment of writing, the generation garbage collection feature of SpiderMonkey is still experimental.

The allocation time results for this experiment illustrate another GC feature which is orthogonal to generational garbage collection, namely incremental marking of the old generation. V8N performs slightly better for long-living objects than the default V8 setting, except for one data point. Since both V8 and V8N implement the same generational GC policy, the performance difference may be explained by the overhead of incremental marking the old generation. We will discuss this feature in detail in Section 6.2.

Figure 10 shows on the y axis the maximum resident set size in MB. For all VMs longer object liveness increases the size of the heap since the time quantum is kept constant. SM, SMN, and SMG show constant memory consumption for object liveness up to five which suggests that for this range SpiderMonkey allocates more memory from the operating system than it actually needs. The differences in the heap size of different VMs also grow with the liveness of the objects where we observe that incremental marking of the old generation in V8 causes additional memory overhead compared to V8N.

Figure 11 shows on the y axis the average allocation time jitter (coefficient of variation) for this experiment. While SM, SMN, and SMG only show a slightly increasing allocation time jitter, for V8N and V8E the allocation time jitter nearly doubles. This suggests that moving objects from the new generation to the old generation adds to the latency of tracing the old generation as well. V8, implementing incremental marking of the old generation, maintains constant latency for long-living objects.

We demonstrate in this experiment that ACDC-JS is capable of exposing the performance impact of generational garbage collection. Together with the results of our heap analysis we conclude that real applications benefit from faster allocation of small objects with a high probability. We also conclude that for long-living objects the tracing policy of the old generation affects allocation performance. We will isolate this impact in the next section.

Parameter	Value
time quantum	512 KB
deallocation delay	increasing from 10 to 100
benchmark duration	200
access live objects	TRUE

Table 4: Non-default ACDC-JS settings for the latency experiment.

6.2 Capabilities of ACDC-JS: Latency

An important performance characteristic mostly ignored by state-of-the-art JavaScript benchmarking suites is memory management latency. Incremental marking is an approach to reduce latency, e.g., pause times caused by the GC, at the expense of higher memory consumption due to delayed collection. In this experiment we configure ACDC-JS to expose the performance impact and trade-offs introduced by incremental marking.

In order to isolate the impact of incremental marking we configure ACDC-JS to increase the load exclusively on the marking phase. We achieve this by emulating a reachable memory leak (across data points). In particular, we increase the size of the object graph that must be traced by the GC while keeping the impact of generational collection constant (each object is promoted to the old generation only once). ACDC-JS implements a so-called deallocation delay to emulate such a mutator behavior. A deallocation delay of x delays the collection of objects by x time quanta. The non-default ACDC-JS parameters for this experiment are listed in Table 4.

Figure 12 shows on the y axis the average allocation time per time quantum for this experiment. The growing amount of reachable garbage only increases the allocation time slightly because the amount of newly allocated objects is constant per time quantum. Nevertheless, the absolute values of the best performing configuration SMN are only half of the slowest configuration V8E. However, the focus of our evaluation is on exposing the performance impact of certain VM implementation details and not on a qualitative comparison of different VMs although such a comparison is possible with ACDC-JS as well and might be interesting to VM developers.

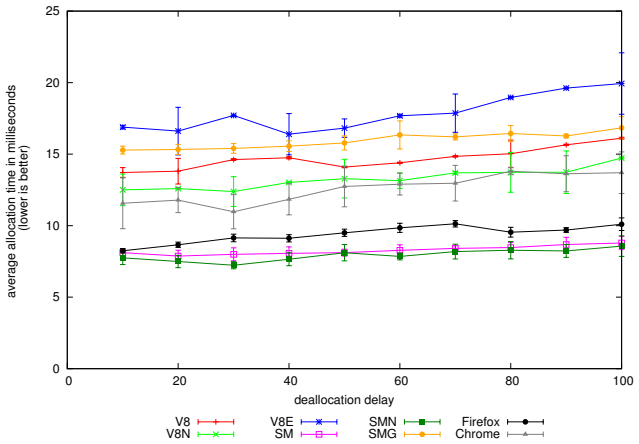

Figure 12: Average allocation time for an increasing deallocation delay.

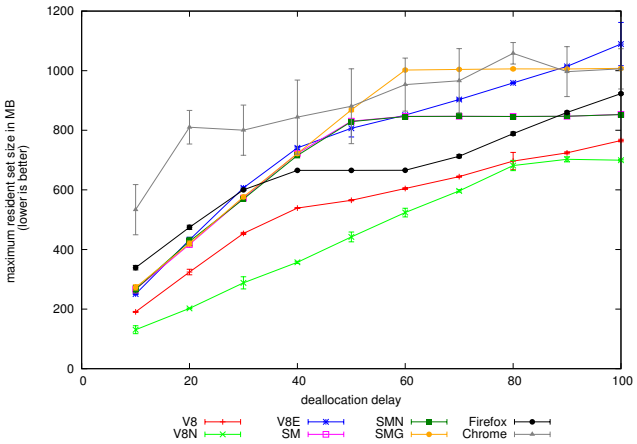

Figure 14: Maximum resident set size for an increasing deallocation delay.

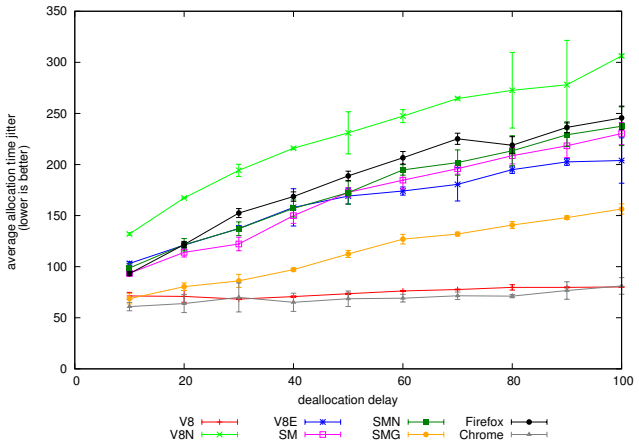

Figure 13: Average allocation time jitter (coefficient of variation) for an increasing deallocation delay.

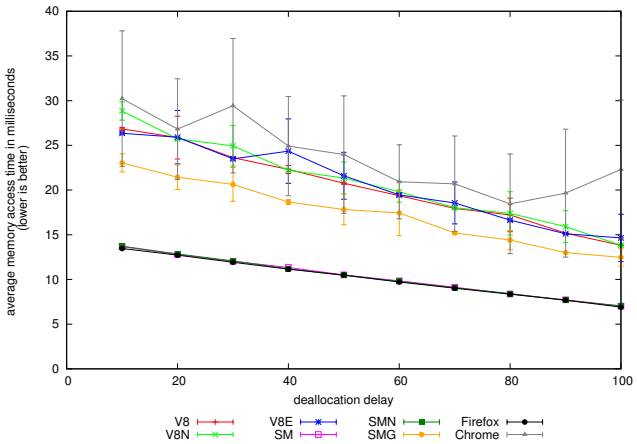

Figure 15: Average memory access time for an increasing deallocation delay.

The key trade-off with incremental marking is allocation latency versus memory consumption. The average allocation time jitter (coefficient of variation) reflecting the variation of allocation latency is presented on the y axis of Figure 13. Here the incremental marking policy of V8 and Chrome show low, constant allocation time jitter. For all other configurations the allocation time jitter increases with an increasing deallocation delay caused by longer GC pause times due to tracing. The incremental marking implemented in SM shows no impact because its integration policy requires event loop actions to trigger incremental marking.

The maximum resident set size for this experiment is shown on the y axis of Figure 14. The growing amount of reachable garbage causes higher memory consumption for all configurations. However, V8N consumes less memory than V8 because V8N reclaims garbage earlier than with incremental marking in V8. This illustrates the trade-off of allocation latency versus memory consumption with incremental marking. SM, SMN, and SMG show constant memory consumption for a deallocation delay larger than 60. This suggests that for smaller deallocation delays they allocate more memory from the operating system than actually required.

Another interesting effect can be observed for the memory access time illustrated on the y axis of Figure 15. We have configured ACDC-JS to also access the live objects after each time quantum. This specifically excludes the objects of the reachable memory leak. As a consequence, the amount of live objects is constant for all deallocation delay settings while the ratio of live objects versus all heap objects decreases with an increasing deallocation delay. The change in ratio may cause the interesting characteristic of faster memory access for larger heaps. Also note the absolute difference in memory access performance for SM, SMN, and Firefox compared to the other configurations. These configurations allow much faster memory access because they do not implement generational garbage collection and therefore do not require read or write barriers to determine the generation of an object.

Figure 16 shows on the y axis the average memory access time jitter (coefficient of variation). Here we observe increasing allocation time jitter at comparable levels for all configurations. This suggests that the growing access time jitter is not caused by VM implementation details but rather by architectural properties of our experimental setup, e.g., the cache architecture.

This experiment demonstrates ACDC-JS's capabilities of exposing the performance impact of incremental marking. The benefit of incremental marking depends on the requirements of the application. Non-incremental marking yields lower memory footprints

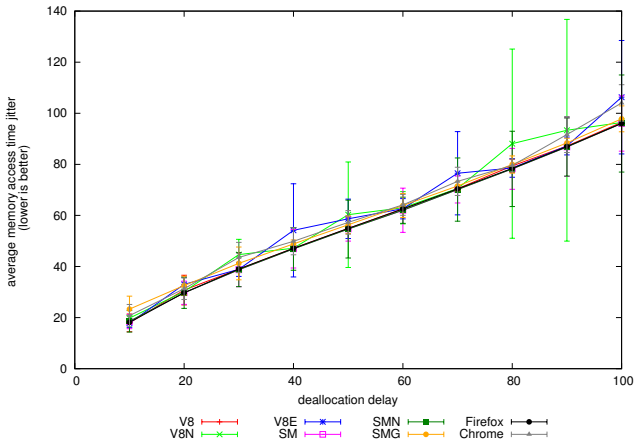

Figure 16: Average access time jitter (coefficient of variation) for an increasing deallocation delay.

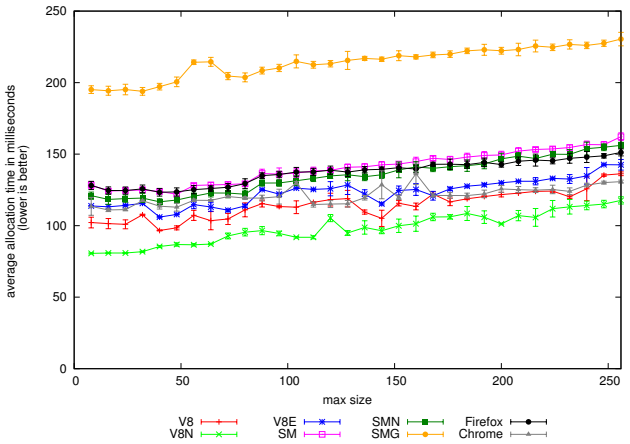

Figure 17: Average allocation time for an increasing object size.

Parameter	Value
min. size	increasing from 8 to 256 B
max. size	min. size
max. liveness	1
time quantum	64 KB * min. size
benchmark duration	50
access live objects	FALSE

Table 5: Non-default ACDC-JS settings for the robustness experiment.

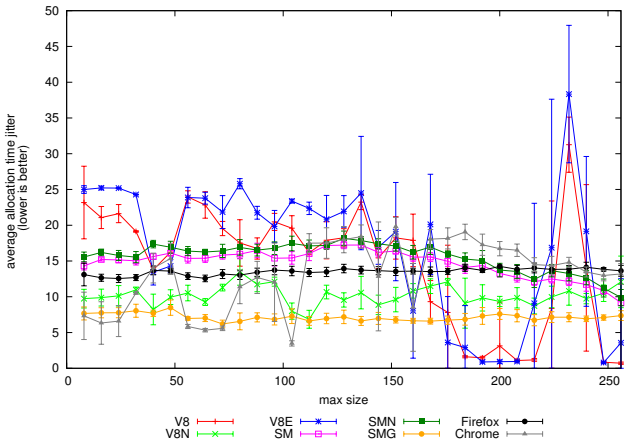

Figure 18: Average allocation time jitter (coefficient of variation) for an increasing object size.

(significant on embedded devices) while incremental marking enables low pause times for soft real-time applications. In addition to the impact of incremental marking, the access time results in this experiment also expose the costs of generation garbage collection. From that we conclude that it is important to explore both, throughput and latency workloads in all relevant performance dimensions to cover all relevant trade-offs in JavaScript VMs.

6.3 Capabilities of ACDC-JS: Robustness

This experiment is intended to expose the impact of object size on allocation performance. Although our heap analysis suggests that small objects occur more frequently it is important for VMs to be robust against a large variety of object sizes. With robustness we mean that continuous variation of a workload characteristic yields continuous response of all performance metrics.

We configure ACDC-JS to allocate a constant number of short-living objects of an increasing size starting at 8 bytes up to 256 bytes to reflect the results of our heap analysis. The number of allocated objects is controlled to be constant by increasing the time quantum linearly to the object size parameters. The non-default ACDC-JS settings are listed in Table 5.

Figure 17 shows the average allocation time for this experiment. All VMs show similar behavior, although at different absolute values. The allocation time tends to increase with an increasing object size suggesting robust behavior for all object sizes.

The allocation time jitter (coefficient of variation) illustrated in Figure 18, on the other hand, shows significant fluctuations for some VMs. SM, SMN, and SMG show continuous behavior of allocation time jitter where SM and SMN even show an improvement for larger object sizes. Running this setting of ACDC-JS on V8 and V8E causes significant outliers for some object sizes which is an unexpected behavior. This demonstrates ACDC-JS's capabilities of exposing performance anomalies for certain workloads by exploring a large set of workload configurations. Furthermore, ACDC-JS can also be used to debug unexpected performance results. In the following, we will give an example of digging deeper into the data collected by ACDC-JS.

The maximum resident set size for this experiment is shown in Figure 19. Again, while most settings for the min. size yield a continuous change in response, this workload triggers outliers in memory consumption for the same workload parameters as for the allocation time jitter. This suggests a relation between the discontinuous behavior of the allocation time jitter and the memory consumption. However, since the maximum resident set size is an aggregation of memory consumption over time, we have to inspect the traces of the resident set size (created by ACDC-JS on the fly) for object sizes that yield outliers in memory consumption.

Figure 20 shows a trace of the resident set consumed by V8E for a min. size of 200 bytes and 232 bytes, respectively, because these settings show large differences in the allocation time jitter of V8E in Figure 18 as well as in the maximum resident set size in Figure 19. The x axis gives the time stamp of the resident set size sample taken during the execution of ACDC-JS. As a consequence, the line for a min. size of 200 bytes is shorter because the execu-

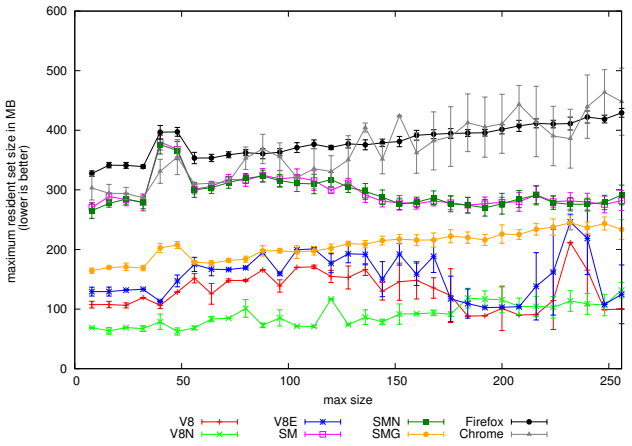

Figure 19: Maximum resident set size for an increasing object size.

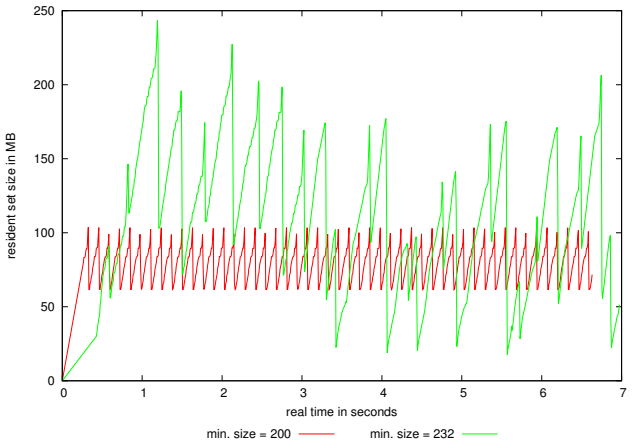

Figure 20: Trace of the resident set size consumed by V8E for a min. size of 200 bytes and 232 bytes.

tion time is shorter. We still consider the graphs comparable for debugging purposes. For a min. size of 200 bytes the VM seems to allocate and deallocate always the same amount of memory from the operating system (through mmapp and munmap system calls) at a constant rate whereas for a min. size of 232 bytes the resident set grows and shrinks at much larger quantities yielding larger workload variations to the operating system and therefore a larger variation of the pause times causing a higher allocation time jitter. This behavior can also be observed for other outliers in Figure 18 suggesting that resource acquisition and release policies work well for some workload settings but do not work as expected for other workloads. By exploring the configuration space of ACDC-JS such workload settings can be found whereas other benchmarks might never trigger such a performance problem.

This experiment demonstrates ACDC-JS capabilities of exposing robustness issues by observing discontinuous responses for continuous workload variations. By analyzing the data collected by ACDC-JS before aggregation we are able to narrow down the causes of irregularities. In the case presented in this experiment we suspect the controller regulating the allocation and deallocation of address space from the operating system to become unstable for certain workloads. We have already informed the V8 development team which also considers this behavior a performance bug.

Parameter	Value
max. liveness	16
time quantum	64 KB
benchmark duration	20
access live objects	FALSE

Table 6: Non-default ACDC-JS settings for emulating the size and liveness behavior of an average application.

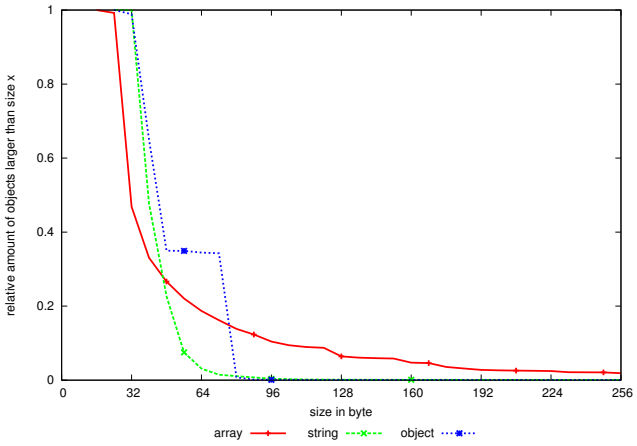

Figure 21: Size distribution of ACDC-JS emulating real application behavior, cf. Figure 2.

We have also experimented with the heap structure properties root distance and number of references but changing these parameters did not expose interesting results. Also for space reasons, we omit these results. However, for future parallel tracing GCs the heap structure properties might affect the tracing performance. We leave a detailed exploration of these properties to future work.

6.4 Capabilities of ACDC-JS: Emulation of Real Workloads

ACDC-JS does not intend to emulate one specific workload but rather the average allocation behavior of real applications. We believe that emulation of average allocation behavior together with ACDC-JS's capabilities to also explore corner cases (as presented in the previous experiments) covers a larger range of possible workloads than emulating one specific real application.

We verify experimentally that the allocation behavior implemented in ACDC-JS corresponds to the results obtained in our analysis of real web applications. For this purpose we apply the ACDC-JS configuration listed in Table 6 but do not change any parameter during the experiment. Instead, we apply the snapshot technique described in Section 3 to generate the distributions for object size and object lifetime. We report size and lifetime distributions for arrays, strings, and objects only. Currently, ACDC-JS does not support any other types. We omit the distributions for the graph properties out-degree, in-degree, and root distance here since we could not observe significant differences in VM performance when changing these properties as mentioned above.

Figure 21 illustrates the size distribution of arrays, strings, and objects allocated by ACDC-JS. The scales of the x and y axes are the same as in Figure 2 to enable visual comparability. We observe a similar distribution of object sizes in ACDC-JS as seen on average in the 13 real web applications. Note that the "knee" in the graph for objects is caused by ACDC-JS's management

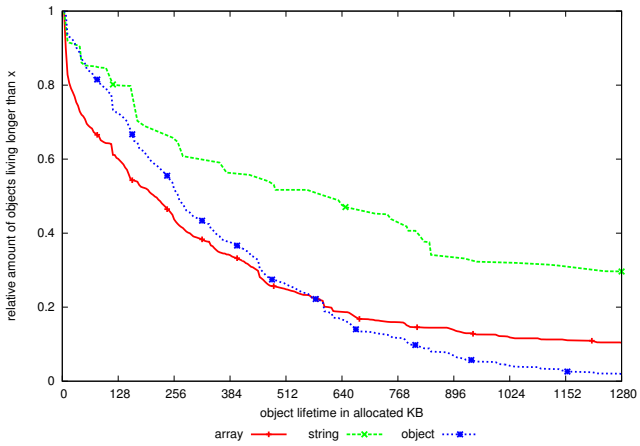

Figure 22: Lifetime distribution of ACDC-JS emulating real application behavior, cf. Figure 3.

data structures. We have nevertheless not observed any impact of ACDC-JS's management data structures on VM performance.

Figure 22 shows the lifetime distribution of arrays, strings, and objects allocated by ACDC-JS. Again, the scales of the axes are the same as in Figure 3 for visual comparability. The distribution we observe with ACDC-JS is again similar to the average distribution obtained from the real web applications in Section 3. For strings, however, we observe a slightly longer lifetime with ACDC-JS because, unlike objects and arrays, strings in ACDC-JS are allocated by slicing and concatenating parts of one large static string. As part of our future work we intend to investigate different string allocation strategies like the String.fromCharCode() method. However, since the distributions are already very similar we do not expect significant changes in the results of the experiments described above.

7. Conclusions

We have presented ACDC-JS, an open source JavaScript benchmarking tool that implements a configurable mutator based on the analysis of real web applications. The tool can be configured to approximate average mutator behavior but also cover corner cases of JavaScript memory management workloads. We have also extended existing work on empirical JavaScript heap analysis of real web applications with object size and heap structure distributions.

Our experimental evaluation shows that ACDC-JS is capable of exposing significant differences in the performance of state-of-the-art JavaScript virtual machines. The evaluation also highlights performance trade-offs that cannot be captured with existing benchmarking suites. As part of future work we plan to investigate allocation behavior of multi-threaded managed languages and to explore ACDC-JS's capabilities for fuzz testing garbage collectors. At the point of writing, ACDC-JS is already in use by the V8 developers for benchmarking GC latency.

Acknowledgments

This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23) and Google, Inc. (Project ACDC4GC).

References

[1] Are We Fast Yet?, 2014. URL `https://github.com/haytjes/arewefastyet`.

[2] Date.now, 2014. URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now`.

[3] Chrome DevTools, 2014. URL `https://developers.google.com/chrome-developer-tools`.

[4] Kraken, 2014. URL `https://wiki.mozilla.org/Kraken`.

[5] Octane 2.0, 2014. URL `https://developers.google.com/octane`.

[6] PerfMeasurement.jsm, 2014. URL `https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules/PerfMeasurement.jsm`.

[7] Performance.now, 2014. URL `https://developer.mozilla.org/en-US/docs/Web/API/Performance.now()`.

[8] SeleniumHQ Browser Automation, 2014. URL `http://docs.seleniumhq.org`.

[9] SunSpider 1.0.2 JavaScript benchmark, 2014. URL `http://www.webkit.org/perf/sunspider/sunspider.html`.

[10] Web Workers, 2014. URL `http://dev.w3.org/html5/workers`.

[11] M. Aigner and C. M. Kirsch. ACDC: Towards a Universal Mutator for Benchmarking Heap Management Systems. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 75–84. ACM, 2013.

[12] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[13] B. Livshits, P. Ratanaworabhan, D. Simmons, and B. G. Zorn. JSMeter: Characterizing Real-World Behavior of JavaScript Programs. Technical report, Microsoft Research, 2009.

[14] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10. USENIX Association, 2010.

[15] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12. ACM, 2010.

[16] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 677–694. ACM, 2011.

[17] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78. Springer-Verlag, 2011.

[18] B. Zorn and D. Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Not.*, 27(12):71–80, Dec. 1992.