Bachelor Thesis

Observation of Real-World Web Applications to Obtain an Realistic JavaScript Heap Behavior

Mario Preishuber
mario.preishuber@cs.uni-salzburg.at

Department of Computer Sciences
University of Salzburg
Austria

February 27, 2014

Advisor

Professor Christoph Kirsch
ck@cs.uni-salzburg.at

Contents

List of Figures

Abstract

Improving the performance of a JavaScript virtual machines is a hot topic. There are industry-standard benchmarks suits to support implementation and optimization of JavaScript virtual machines, but studies, like [1], illustrate that these benchmarks do not represent a real-world web application behavior. Since JavaScript is a fully garbage collected programming language improving performance of the memory management of a JavaScript virtual machine may improve performance in general. To be able to improve performance it requires better understanding performance deficiencies. To optimize a JavaScript virtual machine a configurable and realistic benchmarking of memory management is needed. Realistic benchmarking is only possible if typical JavaScript heaps are known. This requires intensive analysis of JavaScript heaps of real-world applications.

We propose our analysis results of JavaScript heaps of real-world applications. Our analysis results will support developer to implement realistic benchmarking suits which are able to simulate realistic JavaScript heap behavior. We observe 11 popular real-world web applications. Depending on the observed application we executed different user interactions. We use a sampling mechanism to generate frequently a snapshot of the current JavaScript heap. We analyze these snapshots about structure and distributions of object properties.

This thesis present an analysis of popular and JavaScript-intensive real-world web applications to obtain realistic distributions of object properties and heap structure properties.

1 Introduction

JavaScript is part of many modern web applications and became to an essential component of modern web development. Furthermore, JavaScript enables a dynamic, individual behavior of a web application what is important for the user experience. To realize such JavaScript intensive web applications it is necessary that a browser provides a well performing JavaScript virtual machine. Since JavaScript is a fully garbage collected programming language, like JAVA, the memory management is a task of the virtual machine. As a consequence, the performance of JavaScript virtual machines may be improved by improving their memory management performance. To be able to improve performance it requires better understanding performance deficiencies.

There are industry-standard benchmarks suits to support implementation and optimization of JavaScript virtual machines, but studies, like [1], illustrate that these benchmarks do not represent a real-world web application behavior. To optimize a JavaScript virtual machine a configurable and realistic benchmarking of memory management is needed. Realistic benchmarking is only possible if typical JavaScript heaps are known. This requires intensive analysis of JavaScript heaps of real-world applications.

We propose our analysis results of JavaScript heaps of real-world applications. Our analysis results can be used to implement a realistic benchmark which is able to simulate a real-world JavaScript heap. We observe 11 real-world web applications which are similar to those of [1]. To extract information about the JavaScript heap we used a sampling mechanism which frequent generates a snapshot of the current JavaScript heap. This snapshotting functionality is integrated in Google's Chromium [2] virtual machine V8 [3]. Depending on the observed real-world application we perform different user interactions, the complete list is presented by Table 1. During executing such an user interaction snapshots a generated. Section 2 describes in detail the tool chain we used to generate and analyze snapshots. We analyze these snapshots about structure and distributions of object properties. Our analysis is separated in three parts.

The first one, Section 3, analyzes only the mutator heap. A mutator is the common word of garbage collector people for any possible application. In this section we only observe a subset of objects which are allocated by the mutator and separated them by type.

The second part, Section 4, takes a closer look at the system heap. There are special object types which are allocated by the virtual machine. All objects of these special type represent the system heap. These objects are also part of a snapshot this allows us to use the same metics as in Section 3. We separate the objects again by type.

The third part, Section 5, compares the mutator and the system heap. This analysis present differences and equalities of mutator and system object distributions. In this section the objects are not separated by there type instead we distinguish system and mutator objects. Our analysis of this section illustrate the overhead which is produced by V8.

In this work we extend studies on the allocation behavior of real JavaScript web applications [1]. This thesis makes the following contribution: An analysis of popular and JavaScript-intensive real-world web applications to obtain realistic distributions of object properties and heap structure properties.

2 Analysis Tools

2.1 Simple, artificial mutator

We started our research with some simple, artificial mutators. Our aim was to get an overview of the behavior of different JavaScript garbage collectors. Therefore we used three different mutators, which

- Only allocate short living objects,
- Only allocate long living objects, and
- A combination of the two above

With these three mutators we started our measurements. We differ between so-called *blackbox* and *whitebox* data. The *blackbox* data cover the execution time of a mutator and the real memory (resident set size) used of a mutator process. The *whitebox* data require information about the JavaScript virtual machine. We

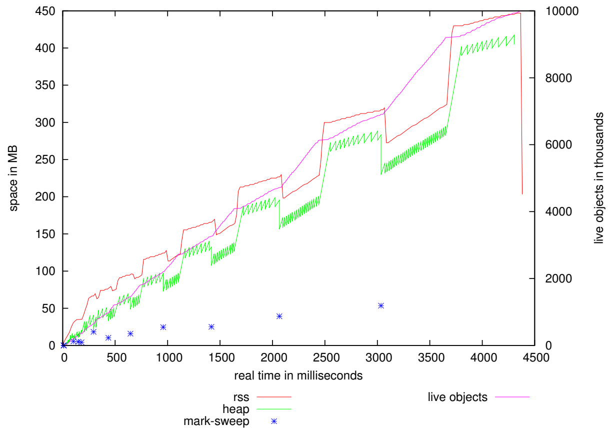

Figure 1: Simple mutator which only allocates objects and never deallocates.

get these information from Google's V8. These data contain information about heapsize, garbage collector frequency, and amount of memory which is collected (in byte).

Figure 1 presents measures of such a simple mutator. This mutator only allocates objects and keeps them live until the mutator terminates. The y-axis on the left shows the memory in MB and is used for the heapsize (heap), resident set size (rss), and mark and sweep (mark-sweep). The y-axis on the right shows the number of live objects in thousands and on the x-axis the real time is displayed. Live objects are objects which are not collected by the garbage collector yet. For the live objects we expect a linear growing, but the Figure shows that the allocation of objects sometimes pauses which is a result of garbage collection. If we look at the resident set size and the heapsize we see the dependency between these. The heapsize presents more details about the garbage collection. The little spikes represent the collection of short living objects. In our case this behavior shows a collection of system objects, because the mutator does not deallocate any objects. The bigger negative going flanks represent a mark and sweep phase of the garbage collector. A mark and sweep phase shows the collection of long living objects. This simple mutator allows us to show some of the characteristics of a garbage collector.

Figure 2 presents the execution time of a mutator which frequently deallocates a static amount of memory. We increase the lifetime of the allocated objects and measure the execution time. The lifetime is the duration since a object is allocated until it is deallocated. The so-called liveness of a object is similar to the lifetime. The x-axis of the Figure shows the maximum liveness of the allocated objects and on the y-axis the execution time is displayed. A comparison of Google's V8 with Mozilla's SpiderMonkey shows significant differences. If we look at the line of V8 we see some big spikes at a liveness of 30, 60, and 90. A reason for these spikes is the communication with the operating system. If the heap size of V8 overruns one of the internal bounds the process requests more memory from the operating system. If the heap size falls below such a bound the process responses memory to the operating system. In our case the heap size is toggling around such bounds. As a result there is a lot more communication with the operation system than usually which has a large impact on the execution time of the mutator.

The conclusion of our first mutator measurements is, that we are able to visualize characteristics of a garbage collector. Furthermore, we can show differences between different virtual machines. These mutators represent some corner cases, and if we think of an realistic JavaScript application there will be a different behavior.

2.2 Tools for obtaining a realistic heap model

Developing a benchmark which is able to simulate a realistic JavaScript heap requires understanding the heap of real-world applications. As shown by some related work [1, 4, 5] the state-of-the-art benchmarks do not represent realistic heap behavior. In this section we describe our analysis to obtain a realistic heap model.

Our analysis are based on frequent generated snapshots of the JavaScript heap of a real-world application.

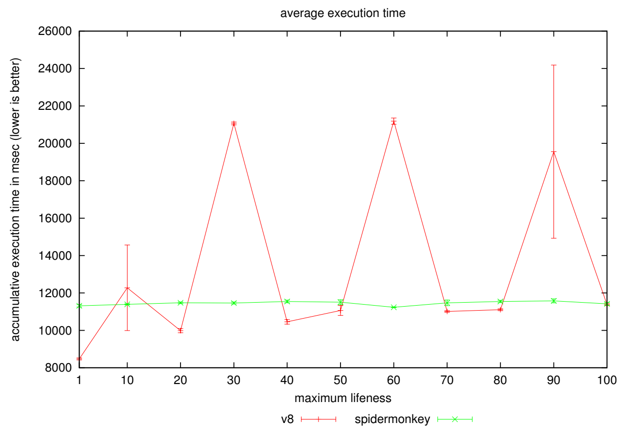

Figure 2: Execution time: V8 vs. SpiderMonkey

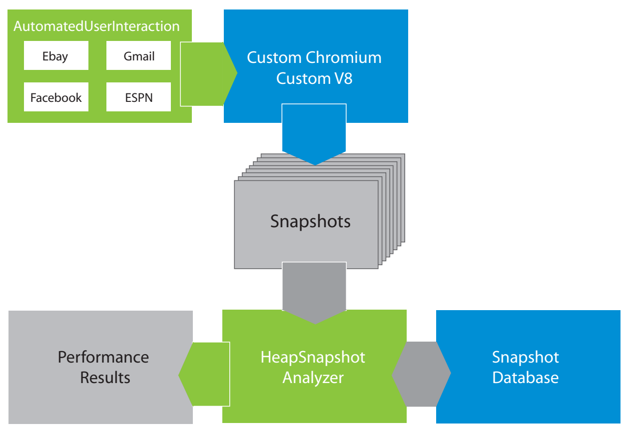

Figure 3: Tool chain to analyze the JavaScript heap of a real-world web application

The list of analyzed applications and proceeded user interactions is shown in Table 1 on page 4. Figure 3 presents the complete toolchain we used to generate and analyze these snapshots.

For the simulation of a realistic user interaction on a real-world application we used Selenium [6]. All analyzed user interactions are collected in so-called `AutomatedUserInteraction` tool (see Section 2.2.1). For execution of a interaction we used a custom version of Chromium. We added a mechanism which periodic produces a snapshot of the JavaScript heap (for details see Section 2.2.2). These snapshots are analyzed by the so-called `HeapSnapshotAnalyzer` which is explained in Section 2.2.3.

2.2.1 AutomatedUserInteraction

To simulate a realistic user interaction we decided to automate this process by using Selenium [6]. A benefit of using Selenium is that we are able to repeat a user interaction as often as we want in exact the same way. Selenium also enables the opportunity to use different browsers. This feature allows us to run a user interaction on different browsers or use our own browser binary in a very easy way. For the heap snapshot generation we used our own customized Chromium binary (for details see Section 2.2.2). The Table 1 shows all these real-world applications and the user interaction we performed.

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 for the 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.

2.2.2 Custom Chromium

Obtaining a realistic JavaScript heap model requires information about objects and structure of a heap of a real-world JavaScript application. As explained in [1] modifying the JavaScript engine of a browser is a good opportunity to get accurate information about the JavaScript heap. We used Chromium [2] the open source project of Google Chrome [7] to capture information about the JavaScript heap.

The Google DevTools [8] provide a functionality to generate heap snapshots. The problem is there is not opportunity to periodic generate such a heap snapshot. We extended the V8 and implemented such a mechanism which also stores the generated snapshots at a given directory. This mechanism works by using the Google V8 API function `TakeHeapSnapshot`. These snapshots contain some meta information, a nodes array, a edges array and a strings array. The nodes array and edges array represent a directed graph, the heap. The strings array is referenced by the nodes and edges array and contains only names. The complete list of properties of a node in the nodes array is shown in Table 4 and the properties of the edges are shown in Table 5. To control the generation of heap snapshots, we extended the V8 with following flags:

- `automatic heap snapshots`
- `heap snapshot interval`
- `heap snapshot prefix`

The `automatic heap snapshots` flag enables the generation of snapshots in general, default it is `false`. To control how frequent snapshots are generated the `heap snapshot interval` flag is used, default value is 64KB. Every time `n` KB are allocated a snapshot is taken. The third flag, `heap snapshot prefix`, is only a prefix of the generated snapshot files, default value is *snapshot*. The produced snapshots are numbered consecutively until the `automatic heap snapshots` flag is set to `false` or the mutator terminates. The Extension of such files it *.heapsnapshot*.

2.2.3 HeapSnapshotAnalyzer

The steps before describe how we capture data about a realistic JavaScript heap. Now we want to explain how we analyzed the generated snapshots.

For each real-world application, listed in Table 1, we used a sample rate of 4KB, i.e., a snapshot is created whenever 4KB of new objects are allocated by the application. The number of generated snapshots depends on the amount of memory which is allocated by a real-world application and the runtime of a user interaction. When a heap snapshot is taken the garbage collector is automatically called. As a result a snapshot only contains live objects.

However, we decided to use a PostgreSQL [9] database, version 9.3, to analyze the snapshots. To handle the preparation of a snapshot for the database as well as the analysis we developed the so-called `HeapSnapshotAnalyzer` which is a Java application. For a realistic heap model, we are interested in the distribution of object types, sizes and lifetimes, the number of outgoing edges, and the distance from a GC root to a object. A heap snapshot represents all live objects of the JavaScript heap and each object has a type. The complete list of object types is shown by Table 7. An object also has a size, which represents the amount of byte the object needs on the heap. The number of outgoing edges represents the number of references an object holds. A snapshot contains special nodes of type synthetic which represent GC roots. These nodes has are used to compute the root distance. The root distance represents the minimum number of edges to reach from a GC root a node. How we compute these properties is described in Section 3.

3 Mutator Heap Analysis

In this section we describe how to some interesting heap properties and present our analysis results of the mutator heap. For our analysis we only care about object types which can be allocated by a mutator. These object types are arrays, strings, user-defined objects, code, closures, regular expressions (regexp), and numbers. We analyzed the distribution of object types, sizes and lifetimes, the number of outgoing edges, and the distance from the GC roots to the objects. The type information is part of a heap snapshot. The distribution of object types is computed over all workloads, i.e., all snapshots of all real-world application.

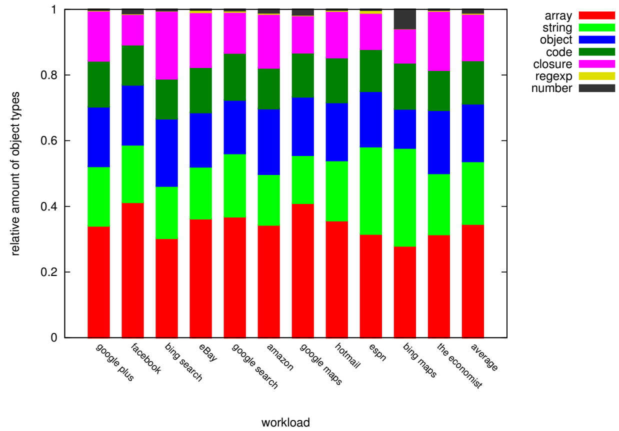

Figure 4: Histogram of the object type distribution for all workloads, confirms [1].

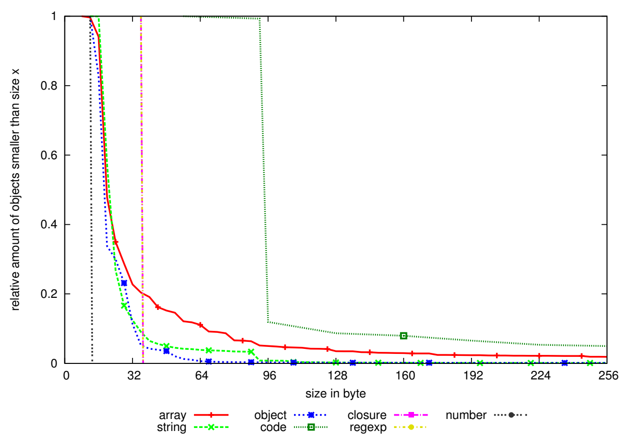

Figure 5: Size distribution of real web applications, confirms [1].

Figure 4 presents the distribution of object types as a histogram. A heap snapshot contains not only objects with a type presented in the histogram. There are several V8 specific types like hidden classes which describe a internal representation of object properties, and GC roots have the type synthetic. We do not care about the V8 specific types, to be platform independent. Our analysis show that the most objects are of type array, string, or user-defined object. The dominance of arrays differs from the results in [1]. A reason for this behavior is if it is necessary to a copy a array because it grows to much, we treat the new array as a logically new heap object. This behavior results in an increasing number of objects and the average lifetime decreases.

Figure 5 shows the object size distribution for each type. The x-axis displays the object size in byte and the y-axis shows the relative amount of objects living shorter than x. Each object on the JavaScript heap is identified by a unique V8 internal id which is a property of a heap snapshot. The size of a object is also a property of a snapshot. This allows us to count objects of the same size of all snapshots of all workloads. As the Figure shows depends the object size distribution on the type of the objects. There are few large object and a lot of small objects. This behavior is similar to the allocation behavior of C programs which is shown in [10]. The Figure also shows that there are object types with a fix size, e.g., numbers with a size of 12 byte.

As explained above it is possible to identify an object by an unique id which never changes. This fact allows us to compute the lifetime of an object. Figure 6 illustrates the distribution of the lifetime of all snapshots of all workloads separated by the object type. The lifetime is measured in allocated bytes which are allocated during a object is live. As a result of the sampling rate of the snapshot generation which is 4

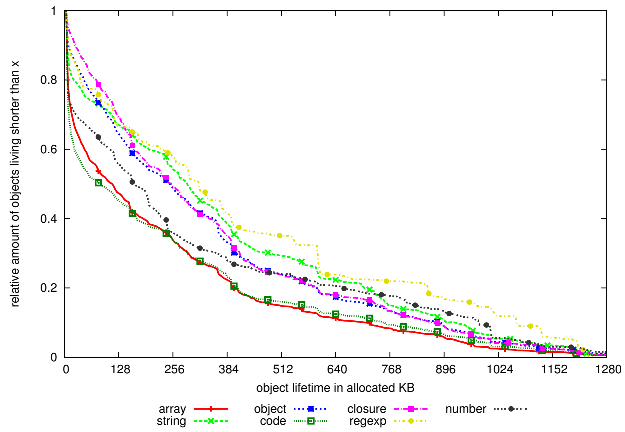

Figure 6: Lifetime distribution of real web applications, confirms [1].

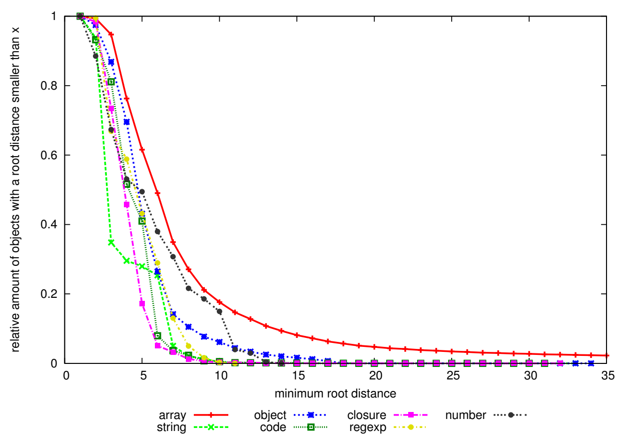

Figure 7: GC root distance distribution of real web applications, confirms [1].

KB, the minimum lifetime of a object is also 4 KB. These amount of alloced byte is displayed on the x-axis and on the y-axis is the relative amount of objects living shorter than x presented. The Figure shows that the lifetime of arrays is significant smaller the lifetime of strings. What results of the fact, if a array has to be copied, because of growing to much, a logically new array is created. This approach influences the liveness of arrays, but from the garbage collector perspective this way of counting is okay, because the old array also has to be deallocated.

An important property of a heap structure is the depth of a heap graph. So we computed the minimum distance of a node to it's GC root. A heap snapshot contains special nodes which represent GC roots, these nodes are of the type synthetic. We use a recursive depth-first search algorithm and compute the root distance for each node of all snapshots of all workloads and separated again by type. The x-axis of Figure 7 shows the minimum root distance and the y-axis presents the relative amount of objects with a root distance smaller than x. This Figure shows that the most objects have a small root distance.

Another interesting property by talking about the structure of a heap is the number of outgoing edges of a node. The number of outgoing edges, the so-called out-degree, results of the fact that the snapshots represent a directed graph. So this information is provided by the heap snapshots. Figure 8 shows the distribution of the out-degree separated by the object type. The out-degree is computed for each node of each snapshot of all workloads. The x-axis of the Figure presents the out-degree and on the y-axis is the relative amount of objects with an out-degree smaller than x displayed. Arrays have a low out-degree what indicates that most arrays contain primitive types.

If we talk about the out going edges of a node we want also to know how the number of ingoing edges

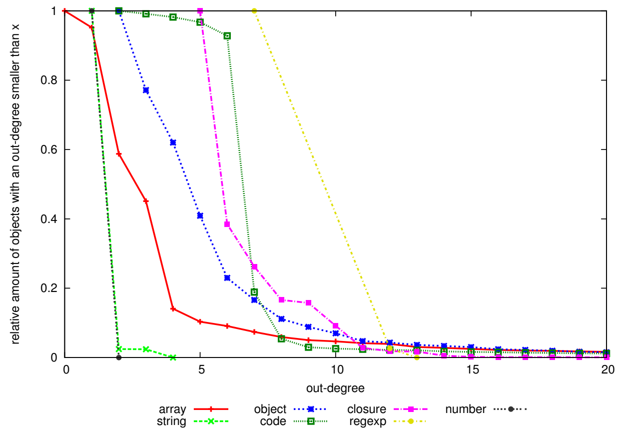

Figure 8: Out-degree distribution of real web applications, confirms [1].

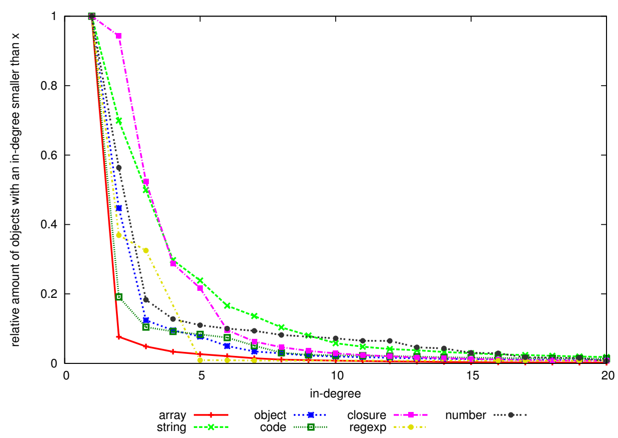

Figure 9: In-degree distribution of real web applications, confirms [1].

looks like. The number of ingoing edges, the so-called in-degree, is not an explicit property of a snapshot, i.e., it requires some computations. In other words the in-degree represents the number of parents a node has and this is the way how it is computed. Before inserting a snapshot into the database some preparations are required and during these preparations the number of parents of a node is counted. Figure 9 shows the distribution of the in-degree over all nodes of all snapshots of all workloads separated by type. On the x-axis is the in-degree displayed and on the y-axis is the relative amount of objects with an in-degree smaller than x is shown. The Figure presents a quite different behavior than Figure 8 which shows the out-degree distribution. The behavior of the different types is more similar that it is for the out-degree. As expected have strings a higher in-degree, because often are strings allocated once and reused.

To complete our analysis of the heap structure we have a look at strongly connected components. We compute the strongly connected components with the Trajan's algorithm [11]. We are interested in the size of a strongly connected component, i.e., the number of edges which the components contains. What we also want to know is the number of strongly connected components of a heap graph. One snapshot represents the current heap, since a heap is a directed graph we analyzed each snapshot separate. The Table 2 shows our analysis results, the values are the average values of all snapshots of all workloads. If we compare the average and the median of the quantity we recognize a significant difference. This comparison shows there are some extreme values, which is also illustrated by the maximum, but the most heaps contain less then 50 strongly connected components. The size of these strongly connected components presents a similar result. There are a lot of small strongly connected components and a few large.

strongly connected components	size	quantity
minimum	0.286	0.782
maximum	53,729.857	285.916
average	5,316.409	11.177
quantile 25	2,304.786	3.107
quantile 50	3,082.429	6.329
quantile 75	3,629.714	14.655
quantile 90	12,199.429	36.211
quantile 95	25,979.000	49.448

Table 2: Summary of the measurements of strongly connected components.

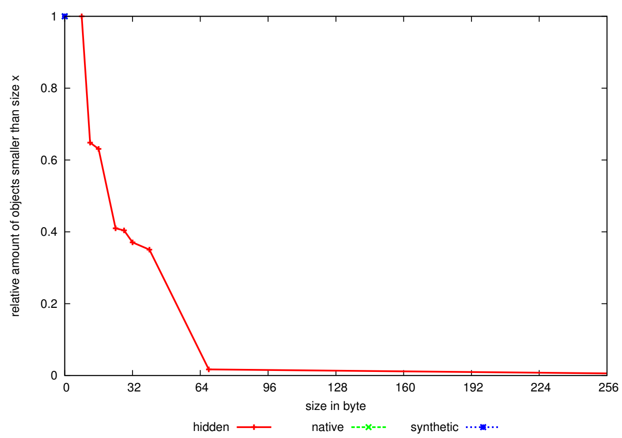

Figure 10: Size distribution of real web applications.

4 System Heap Analysis

In this section we present our analysis results of the system heap. For our analysis we only care about objects of with type hidden, native, or synthetic. Objects of type hidden represent hidden classes. These objects are generate by Google's V8 and represent the properties of a object. GC roots has the type synthetic and objects of type native represent other objects, i.e., DOM roots. We analyzed the same properties as described in Section 3 in the same way.

The result of the object type distribution of the system types showed that about 99% of these objects are of type hidden for all workloads. There extreme few objects of the types synthetic and native. One reasons is that objects of type synthetic represent GC roots and it is plausible that there are not much such objects. Nevertheless, we care about them, because of completeness.

Figure 10 illustrates the size distribution of the system object types of all snapshots of all workloads. The size of native and synthetic objects is always zero in a heap snapshot. This is a special behavior and represents not the real size of these objects. Nevertheless, is this Figure interesting because is shows the size distribution of hidden classes. The Figure shows on the x-axis the size in byte and on the y-axis the relative amount of objects smaller than size x. We see that there are only few objects of type hidden with a size over 64 byte. It seems that the size of 64 byte is some kind of bound, because there is a hard back.

In Figure 11 we present the lifetime distribution over all snapshots of all workloads separated by the object type. The x-axis shows the object lifetime in allocated KB. We decided to use this metric to get rid of the lifetime in snapshots representation. The sample rate of the snapshot generation is 4KB of allocated memory and this leads to an minimum lifetime of 4KB. The lifetime of hidden classes shows a similar behavior than the lifetime of mutator objects of type object as explained in Section 3, especially Figure 6 on page 7 illustrates.

The root distance of nodes in a graph is an interesting parameter of the graph characteristic. If we look at the root distance of system objects we are faced with some special cases, because objects of type synthetic

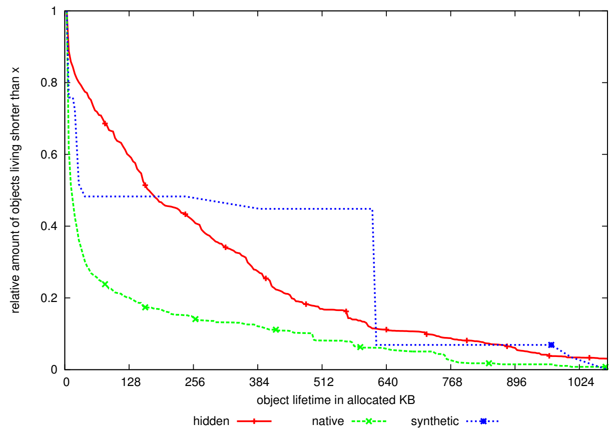

Figure 11: Lifetime distribution of real web applications.

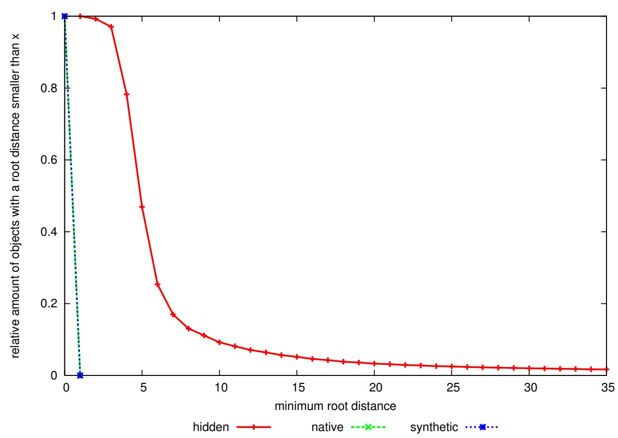

Figure 12: Root distance distribution of real web applications.

represent GC roots which are used to compute the root distance. It is possible that a object of type synthetic also has a root object. So the root distance of synthetic objects is always zero or one. Objects of type native show a similar behavior because these are also used to represent DOM roots. Nevertheless, the behavior of objects of type hidden provides interesting informations. Figure 12 illustrates the root distance distribution of system objects. On the x-axis of the Figure shows the minimum root distance and on the y-axis the relative amount of objects with a root distance smaller than x. This Figure shows that hidden classes have a small root distance and that there are only few objects with a higher root distance.

The Figure 13 presents the distribution of outgoing edges of a node, the so-called out-degree. This illustration shows the results computed over all snapshots of all workloads. In this Figure we see that objects of type hidden have a out-degree of less than ten. If we have a look at the native line, we recognize that this kind of objects have a significantly higher out-degree than objects of type hidden. Especially objects of type synthetic have a extreme high out-degree, because they represent the GC root and we conclude the heap tree is very breadth.

5 System vs. Mutator

In the previous sections we described our analysis results separated in system heap and mutator heap. We presented the differences and similarities distinguished by the type of the objects. In this section we compare system and mutator objects and do not care about the type of the objects. As explained in Section 3 are objects of type array, string, user-defined object, code, closure, regular expression (regexp), or number are

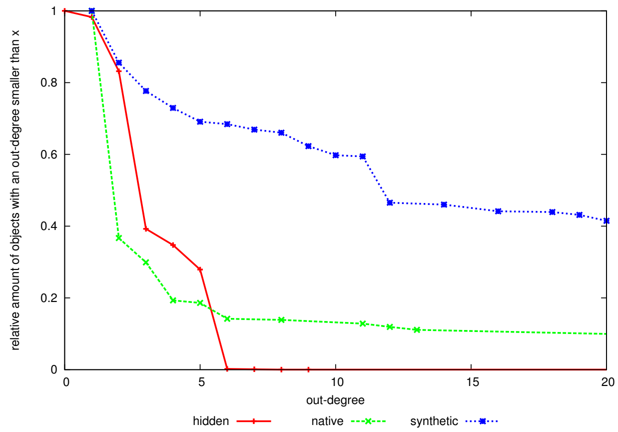

Figure 13: Out-degree distribution of real web applications.

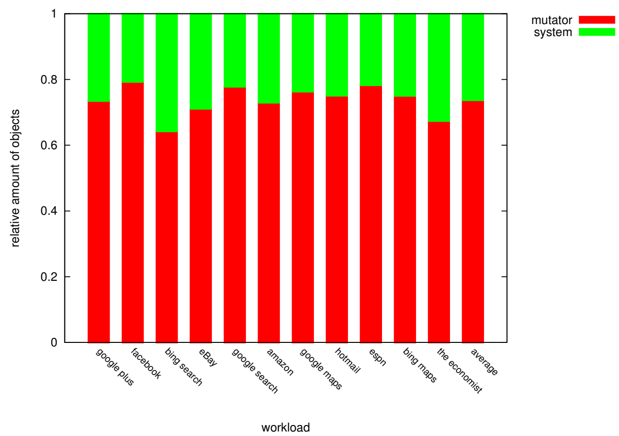

Figure 14: Histogram of the object distribution for all workloads.

mutator objects. Objects of type hidden, native, or synthetic are system objects as described in Section 4. This analysis illustrate the overhead produced by Google's V8.

Figure 14 shows a histogram of the object distribution for all workloads. On the x-axis the workloads are listed and on the y-axis the relative amount of objects is displayed. This Figure shows that there are about 25% of the allocated objects are system objects. As explained in Section 4 most of these objects are of type hidden.

The distribution of the out-degree is presented by Figure 15. On the x-axis is the out-degree shown and on the y-axis the relative amount of objects with an out-degree smaller than x is presented. This Figure shows a quite different behavior of system and mutator objects. The only similarity is that there are few objects with an out-degree higher than six. More detailed, there are less than 1% of system and less than 10% of mutator objects with a higher out-degree than six. It is also interesting that the out-degree of mutator objects is significantly smaller as the out-degree of system objects.

The lifetime distribution, illustrated by Figure 16, presents a similar behavior of system and mutator objects. A reason for this is the dependency of hidden classes and user-define objects and again the fact that most of the system objects are of the type hidden. On the x-axis of the Figure the object lifetime in allocated KB is shown and on the y-axis the relative amount of objects living shorter than x is displayed.

Figure 17 presents the distribution of the root distance of all snapshots of all workloads separated in system and mutator objects. On the x-axis the minimum root distance is shown and on the y-axis the relative amount of objects with a root distance smaller than x is shown. The behavior of the system and mutator objects is extremely similar. We conclude that there might be a dependency of the position of

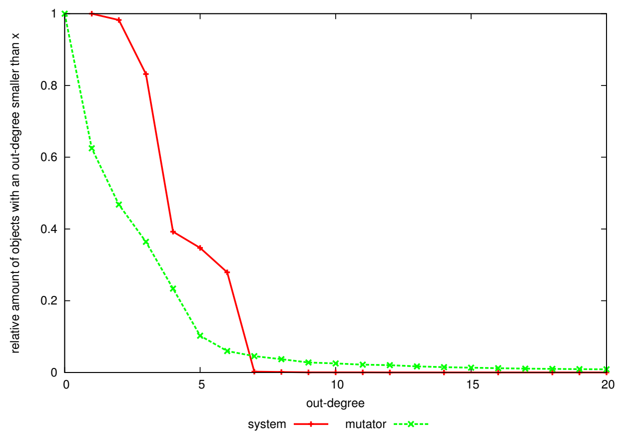

Figure 15: Out-degree distribution of real web applications.

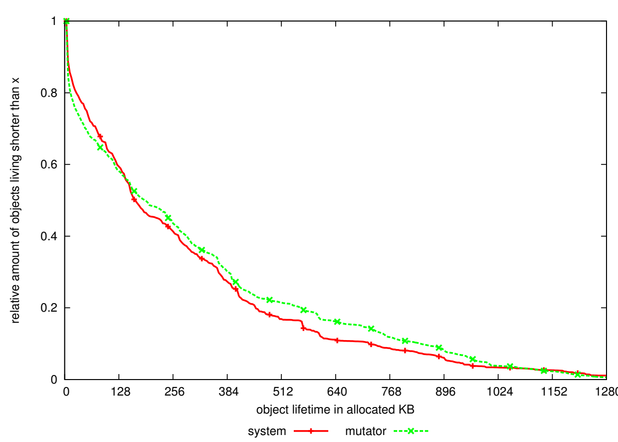

Figure 16: Lifetime distribution of real web applications.

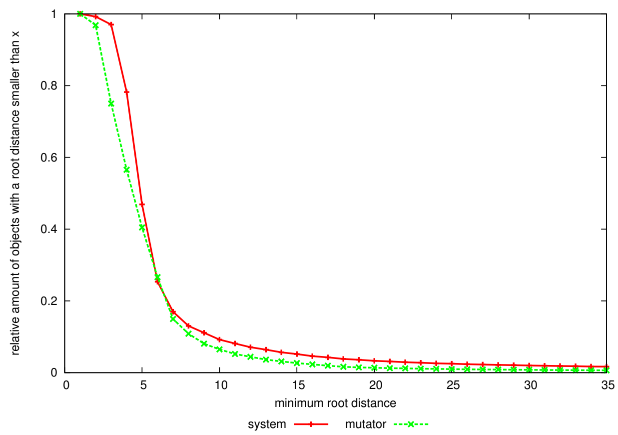

Figure 17: Root distance distribution of real web applications.

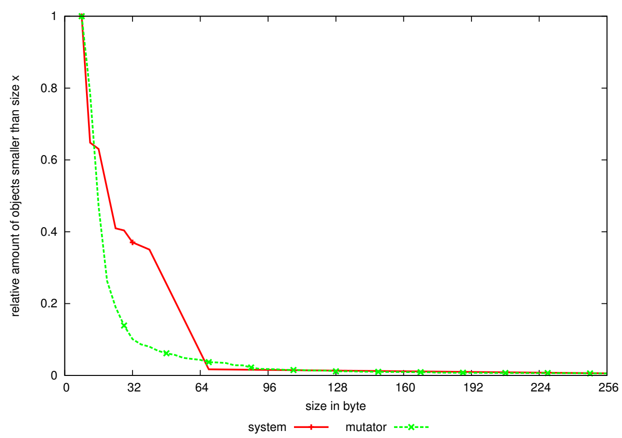

Figure 18: Size distribution of real web applications.

system objects and the position of mutator objects in a heap graph. A reason could be that a user-defined objects has a reference the hidden class which represents the properties of a object.

If we compare the size distribution of system and mutator objects we recognize that system objects are significantly bigger. This behavior is illustrated by Figure 18, which shows on the x-axis the size in byte and on the y-axis relative amount of objects smaller than size.

6 Conclusions

We presented our analysis of popular and JavaScript-intensive real-world web applications to obtain realistic distributions of object properties and heap structure properties. Our analysis support developers by implementing realistic JavaScript benchmarking suits which are able to simulate a realistic JavaScript heap behavior. Such benchmarks will help to improve performance of memory management systems of JavaScript virtual machines. Furthermore, this may be improve the performance of a JavaScript virtual machine in general.

A Appendix

This section presents some additional tables about the generated heap snapshots.

Name	Description
number of nodes	Each snapshot contains a field called `node_count`. The `node_count` field is equal with `number of nodes` and shows how many nodes the heap currently contains. The `nodes_count` field is also an property of the table `snapshots` and exists once per snapshot.
number of edges	This is the same as `number of nodes` only for edges. Each snapshot contains a field `edge_count`. The `edge_count` field is equal with number of edges and shows how many edges the heap currently contains. The `edge_count` field is also an property of the table `snapshots` and exists once per snapshot.
number of leafs	The `number of leaves` is computed as followed: scan all nodes of a snapshot and count those without children. Each node has a property called `edge_count` this value is different from the `edge_count` of a snapshot, only the name is the same. A node has no children if the `edge_count` is zero, because the `edge_count` field contains the number of leaving edges. The `edge_count` field it an property of the `nodes` table.
outdegree	As explained before each node has a property `edge_count` which shows the number of leaving edges. The number of leaving edges is the same as the outdegree of a node, so it follows that `edge_count` is also equal with the outdegree of a node. By using some aggregations of SQL it is easy to compute the maximum, minimum and average number of leaving edges. The `edge_count` field it an property of the `nodes` table.
indegree	The `indgeree` of a node is computed before inserting the data into the database. For each node in the snapshot is counted how many other nodes have a reference to this.
selfsize	The `selfsize` is a property of a node in the snapshot and is computed by the V8 snapshotting mechanism. It shows the size of a node in byte.
rootdistance	The `rootdistance` is an property of the `node` table and is computed for each node before it is inserted in the database. The rootdistance is the shortest path from any root to a node.
strongly connected components	To find strongly connected components in the heap graph the Tarjan's strongly connected components algorithm[11] is used.

Table 3: Heap snapshot metrics

Field name	Description
type	See Table 7
name	A string which names the object. If the object is an mutator variable this name is the same as in the mutator (reference to strings array)
id	A unique id given by the virtual machine (V8)
selfsize	This is the size of the object in byte.
edgecount	Number of reference which leaves the object.

Table 4: Heap snapshot node fields

Field name	Description
type	See Table 6
name_or_index	A string which names the edge. (reference to strings array)
to_node	It is the pointer to the child node. (reference to nodes array)

Table 5: Heap snapshot edge fields

Name	Description
context	A variable from a function context
element	An element of an array
property	A named object property
internal	A link that can't be accessed from JS, thus, its name isn't a real property name (e.g. parts of a ConsString)
hidden	A link that is needed for proper sizes calculation, but may be hidden from user
shortcut	A link that must not be followed during sizes calculation
weak	A weak reference (ignored by the GC)

Table 6: Heap snapshot edge types

Name	Description
hidden	Hidden node, may be filtered when shown to user
array	An array of elements
string	A string
object	A JS object (except for arrays and strings)
code	Compiled code
closure	Function closure
regexp	A regular expression
number	Number stored in the heap
native	Native object
synthetic	Synthetic object, usually used for grouping snapshot items together
concatenated string	Concatenated string (a pair of pointers to strings)
sliced string	Sliced string (a fragment of another string)

Table 7: Heap snapshot node types

References

[1] Benjamin Livshits, Paruj Ratanaworabhan, David Simmons, and Benjamin G. Zorn. JSMeter: Characterizing Real-World Behavior of JavaScript Programs. Technical report, Microsoft Research, 2009.

[2] Chromium, 2014. URL `http://www.chromium.org/`.

[3] V8, 2013. URL `https://developers.google.com/v8/`.

[4] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin 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.

[5] Gregor Richards, Andreas Gal, Brendan Eich, and Jan 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.

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

[7] Chrome, 2014. URL https://www.google.com/intl/en/chrome/.

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

[9] PostgreSQL, 2013. URL http://www.postgresql.org/.

[10] Martin Aigner and Christoph 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.

[11] Tarjan's strongly connected components algorithm, 2013. URL http://en.wikipedia.org/wiki/Tarjan%E2%80%99s_strongly_connected_components_algorithm.