Thinking Inside the Box: Compartmentalized Memory Management

Summary

Varderes Barsegyan
Computer Engineering and Computer Science
California State University, Long Beach
Long Beach, USA
barsegyanvarderes@gmail.com

Rucha Kanse
Computer Engineering and Computer Science
California State University, Long Beach
Long Beach, USA
ruchakanse@ymail.com

***Index Terms*—memory management, heap, optimization**

I. Abstract

JavaScript enables browsers to support dynamic websites and engaging user experiences, independent of the platform in which they are running on. To meet the ever-increasing demands of users across many kinds of devices (e.g. multi-tabbed browsing), compartmentalized memory management has become a priority in JavaScript engines. The main idea presented in this research is to partition the JavaScript heap into small compartments and allocate memory in those compartments based on the domain names of websites. This approach enables garbage to be collected at the compartment level rather than at the level of the entire heap. The researchers show that by implementing a compartmentalized memory management system, the runtime performance of Mozilla Firefox improves by up to 36%, reduces the garbage collection pause times by up to 75%, and decreases the overall memory used by the browser.

II. Motivation & Background

JavaScript performance is a priority due to the highly responsive and dynamic content that must be rendered in browsers. On the other end, the needs and behaviors of the users have evolved. Reports indicate that the average user has about 25 open tabs with some users reaching 200. With these metrics in mind, the researchers set out to improve the V8 JavaScript benchmark points of 50 simultaneously open tabs. Their contributions are summarized as follows:

- The use of compartments in the memory management subsystem
- The identification of good granularity of compartmentalization
- Optimize approach with background finalization and parallel marking
- Evaluation of novel approach

Throughput and pause times of the garbage collection heavily influence the time in which the browser renders content while maintaining efficiency and security. Previous approaches isolate memory management at level of individual tabs. With these approaches, two problems persist:

- *iframe* elements (embedded web pages) require their parent pages to hold references to objects served from different origins. By loading pages into the parent process, the separation of tabbed processes is no longer beneficial.
- Dedicating a process to each tab is memory-intensive, which does not work well with mobile devices with limited memory.

The approach presented here splits the JavaScript based on origin. For example, *http://www.mozilla.com/a.html* and *http://www.mozilla.com/b.html* have the same origin, but *http://mozillalabs.com/a.html* and *https://www.mozilla.com/a.html* do not. Since this method does not rely on processes for isolation, it can work well on a variety of hardware.

III. Design

Prior to compartmentalization, the Firefox scripting engine, SpiderMonkey, partitions the heap into 1MB chunks, which in turn are divided into 4KB arenas. After enough objects are allocated into memory, garbage collection is initiated. The garbage collector then traverses the entire heap to mark objects that are to be deallocated. Traversing the entire heap is wasteful since many unused tabs might not need garbage collection.

A. Compartmentalization

The novelty of the approach presented by the researchers is the grouping of objects in memory based on their origin. Furthermore, these compartments are stable in that they cannot be changes by their encapsulated objects nor can they be split or combined with other compartments. Objects in other compartments are referenced using wrapper objects, while objects within a compartment reference one another directly.

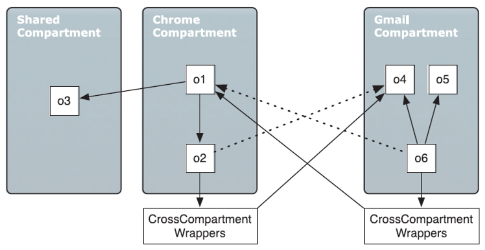


Fig. 3. An overview of possible references between compartments. The dotted arrows represent the old way of communicating between two objects. In the new approach, we add a wrapper object between two objects that reside in different compartments.

Thus, compartmentalization provides the following benefits:

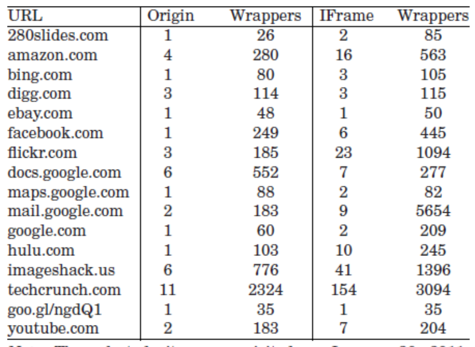
- Garbage is collection one compartment at a time. Areas of the heap not related to a compartment are not walked.
- Wrappers are private to each compartment. Previously they were accessible by functions from other origins. This way, access checks are performed during the creation of wrappers rather than at the time in which they are accessed.
- Object synchronization is no longer required since allocation is done within a compartment.
- The locality of reference is improved since the need for cross-origin referencing is infrequent.

Finally, it is important to note that garbage collection is triggered when the allocation of a compartment reaches a predetermined threshold. A global garbage collection is performed if the overall allocation exceeds 150% of the compartment that initiated the garbage collection. Global garbage collection cannot be avoided because objects in a compartment can cross reference objects in other compartments that must be collected.

B. Granularity of Compartments

Choosing the granularity of each compartment is non-trivial since it is impossible to isolate web programs. On one hand, having very few compartments results in many global walks of the heap. On the other hand, having many compartments will lead to many cross-origin wrappers and compartment metadata. Thus, the researchers performed an experiment at two different granularity levels: separation by origin and separation by *iframe*.

Table I. Compartments and Corresponding Cross Compartment Pointers When Creating New Compartments Per Origin (Columns 2 and 3) or Per `iframe` (Columns 4 and 5)



URL	Origin	Wrappers	IFrame	Wrappers
280slides.com	1	26	2	85
amazon.com	4	280	16	563
bing.com	1	80	3	105
digg.com	3	114	3	115
ebay.com	1	48	1	50
facebook.com	1	249	6	445
flickr.com	3	185	23	1094
docs.google.com	6	552	7	277
maps.google.com	1	88	2	82
mail.google.com	2	183	9	5654
google.com	1	60	2	209
hulu.com	1	103	10	245
imageshack.us	6	776	41	1396
techcrunch.com	11	2324	154	3094
goo.gl/ngdQ1	1	35	1	35
youtube.com	2	183	7	204

Note: The selected sites were visited on January 30, 2011. Some sites required an account in order to perform basic tasks. `goo.gl/ngdQ1` is a shortened link to Google's V8 benchmark suite.

Only eBay benefits from iframe-level separation.

IV. Enhancements

Three important enhancements were made to this approach. The first is *background finalization*, in which the finalization and deallocation of unreachable objects is done is performed using a background thread. Second is *parallel marking*, where the marking of objects for garbage collection is parallelized. Finally, *reducing fragmentation*, where objects with similar lifetimes are allocated near each other. This increases the likelihood of memory chunks being released to the operating system upon termination of a tabs.

V. Evaluation

The researchers evaluated the improvements with respect to runtime performance, memory use, and scalability. Performance measures were done using V8 JavaScript benchmarks created by Google. Space and scalability measurements were done on a Mac Pro workstation with 4GB RAM running Firefox 4.0, and runtime measurements were done on a MacBook Pro with 8GB RAM running Firefox 7. An analysis of the memory usage shows that that peak memory consumption is noticeably higher with compartments by about 13%, but consumption decreases at a faster rate when closing tabs. This was measured by opening 50 tabs in sequence then closing them one at a time.

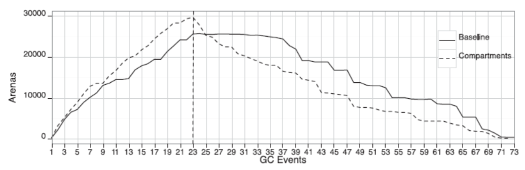
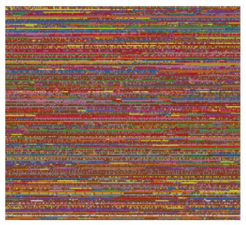
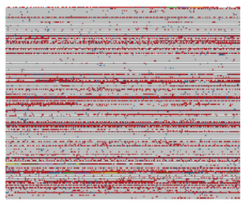


Fig. 5. Opening 50 tabs and closing them again with the baseline and compartment approach. We can see a higher memory consumption peak for the opening process with the new approach, but once we close tabs (to the right of the dotted line), we also deallocate arenas faster.

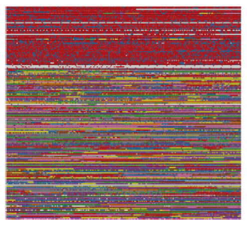
Further analysis of consumption shows a significant reduction in memory fragmentation. In the figure below, 1MB chunks are represented by rows, and 4KB arenas are represented by cells. Cells which are not allocated are white.



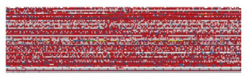
(a) JS heap snapshot for 50 open tabs without compartments.



(b) JS heap snapshot after opening and closing 50 tabs. We perform several GCs after all tabs are closed in order to reclaim all unreachable memory.



(c) JS heap snapshot for opening 50 tabs with the new approach. We see that about the first 30% of the heap is filled with system chunks and the rest with user chunks.



(d) JS heap snapshot for opening and closing 50 pages with the new approach. We see that the system chunks remain but all user chunks are returned to the OS.

Fig. 7. Heaps before and after separating objects based on lifetimes. Colored squares represent 4KB arenas that belong to a certain compartment.

An analysis of scalability shows significant improvements in the time required to open 150 tabs.

Fig. 8. Fragmentation before and after separation of short- and long-lived objects. The dotted line separates the opening and closing of tabs.

Table II. Scalability Test

	Firefox 7	Chrome I	Chrome II	Opera 11.50
`real`	6min 14s	28min 55s	27min 59s	6min 55s
`user`	3min 55s	21min 58s	41min 05s	5min 23s
`sys`	0min 49s	14min 40s	20min 35s	1min 13s

Real time is the elapsed wall clock time in seconds, user time is user mode CPU seconds, and sys time is kernel time CPU seconds.

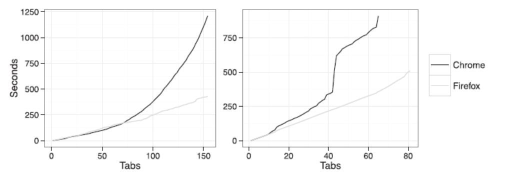


Fig. 9. Wall-clock time required to open 150 tabs in Firefox and Chrome. The graph on the left shows the performance on a high-end laptop with 8 GB RAM whereas the one on the right was obtained on a netbook having only 1 GB of RAM. Note the different scales.

Finally, single-tab and multi-tab performance was analyzed. Single-tab benchmark performance increased by an average of 6% and in some cases 14%. Multi-tab performance increased by an average of 36%. Furthermore, multi-tab garbage collection pause times were decreased by an average of 75%.

Table III. V8 Scores with a Single Tab Running the V8 Benchmark Suite

	Base	Comp	Background Finalize	Parallel Marking	Relative
Richards	8109	8059	8038	8162	1%
DeltaBlue	4954	4813	5139	5020	1%
Crypto	8614	8586	8663	8716	1%
RayTrace	3590	3956	4107	4083	12%
EarleyBoyer	4247	4569	4846	4931	14%
RegExp	2143	2125	2112	2134	0%
Splay	5761	5965	6421	6450	11%
Score	4872	4983	5154	5172	6%

Note: Relative compares the base approach with parallel marking where all optimizations are enabled. Larger scores are better.

Table IV. Average Times Per GC Event for the V8 Benchmark.
With the New Compartment Approach and all Optimizations

	Base	Compartments	Background finalize	Parallel marking	Percent reduction
Marking [ms]	805	474	522	370	54
Sweeping [ms]	1323	1265	236	241	82
Finalize objects [ms]	1072	1051	221	228	79
Finalize strings [ms]	224	200	0	0	100
Total [ms]	2240	1823	837	690	69

Note: We reduce the time spent in the GC during the V8 benchmark by 69%. Relative compares the base approach with the parallel marking approach. Lower values are better.

The websites used to benchmark the 50 tabs can be seen in the online appendix.

VI. Conclusion

The authors have clearly shown performance gains by using their novel memory-management technique. Partitioning the heap in this way improves performance on both desktop computers and mobile devices. In fact, the memory-management techniques presented in this paper are now part of the Mozilla Firefox browser.

As websites become increasingly complex and demanding, there will always be the need to further refine and optimize these algorithms. The authors have set the stage for a new approach to memory management in browsers, and many contributors today are continuously improving these methods.