Compartmental Memory Management in a Modern Web Browser

Gregor Wagner[†§] Andreas Gal[§] Christian Wimmer[†] Brendan Eich[§] Michael Franz[†]

[†]University of California, Irvine [§]Mozilla Corporation

{wagnerg, cwimmer, franz}@uci.edu {gwagner, gal, brendan}@mozilla.com

Abstract

Since their inception, the usage pattern of web browsers has changed substiantially. Instead of sequentially navigating static web sites, modern web browsers often manage a large number of simultanious tabs displaying dynamic web content, each of which might be running a substantial amount of client-side JavaScript code. This environment introduced a new degree of parallelism that was not fully embraced by the underlying JavaScript virtual machine architecture. We propose a novel abstraction for multiple disjoint JavaScript heaps, which we call compartments. We use the notion of document origin to cluster objects into separate compartments. Objects within a compartment can reference each other directly. Objects across compartments can only reference each other through wrappers. Our approach reduces garbage collection pause times by permitting collection of sub-heaps (compartments), and we can use cross-compartment wrappers to enforce cross origin object access policy.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures - Domain-specific architectures; D.3.4 [*Programming Languages*]: Processors - Memory management (Garbage Collection)

General Terms Design, Performance, Experimentation

Keywords Web-Browser Architecture, Isolation, Memory Management, Garbage Collection

1. Introduction

Increasing bandwidth, faster computers, and a JavaScript performance boost over the last few years have enabled web developers to build highly complex web-applications. Browser-based office applications or games can now replace typical desktop applications. This rapid change in the usage pattern of a browser poses a big challenge for browser implementors. The functionality of a modern browser is moving towards the responsibilities usually provided by an operating system. Browser that were once good enough have now become a performance bottleneck. Memory management and garbage collection (GC) is now a severe pottleneck within the browser and especially within the JavaScript virtual machine (VM). Previously, browsing speed was dominated by rendering and network delay rather than GC pause times. Major improvements in rendering and increases in bandwidth solved these old browser issues.

Further, architectural changes like enabling multiple tabs were added to the existing browser code, but the underlying JavaScript execution environment was not redesigned. The implementation of the memory management subsystem does not reflect the high-level configuration of the browser. For example, high level separations such as browser tabs are not reflected in the low-level design of the VM. As a result, web pages loaded in separate tabs encounter many types of interference that affect their memory management, security and performance.

Some web browsers such as Google Chrome or Microsoft Internet Explorer overcome the general separation problem by creating a new process for each new tab or origin. This is good for security since process boundaries act like a "hardware fences" between browsing instances and the memory management can be handled completely separately for every tab. Chrome also copies instances of the JavaScript VM for every process. Since the created browsing instances are not lightweight, they limit the number of processes to 20.

The problem with this solution is twofold: Certain features like browser extensions for common browsers already use a cross-origin communication mechanisms. In order to be backward compatible, Chrome has to collect some origins in one process without any communication restrictions. Furthermore, creating new processes for new origins is not an option for mobile devices. One of the design constraints is that the new approach has to work on the desktop as well as on the mobile version of a browser.

In order to define the problem we look at the previous implementation of the JavaScript heap in Firefox. The JavaScript heap when opening some tabs in the browser is shown in Figure 1. In this example we open some tabs and load popular web pages. The objects are not separated on the heap and it is very likely that all the objects from different origins interleave on the heap. A Facebook object could be right next to a CNN object for example. We also load the V8 benchmark page in order to run the JavaScript benchmarks. Interleaving objects created by benchmark pages with other objects show the drawbacks of the previous implementation:

- *Bad locality*: Objects that are often accessed at the same time are not grouped together.
- *No partial GC possible*: During a GC event, every single object has to be accessed.

Our research proposes a new layer of abstraction for the JavaScript heap. We split the JavaScript heap into sub-heaps which we call compartments. JavaScript objects that are allocated from a

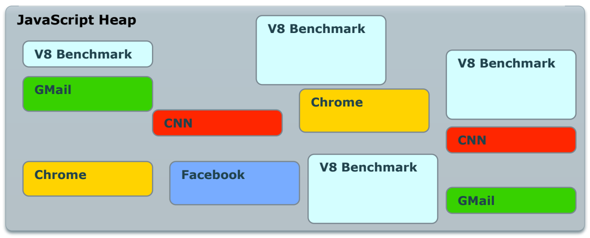

Figure 1. The previous implementation allows object of different origins to be allocated in the same memory region.

certain origin are now placed into the compartment that is associated with the origin. This new abstraction level allows us to:

- Separate memory,
- Improve cache behavior, and
- Perform partial GC and therefore reduce GC pause time.

We implement our research in the open source web browser Firefox [13]. Firefox has about 400 million daily users with market share between 25% and 30% according to [21].

Having many open tabs is not unusual any more. User reports that are collected at Mozilla show that some power users have 200+ open tabs. Running benchmarks in such an environment used to show drastic performance impacts. For example, the V8 benchmark score drops from 4511 to 3017 when only 50 tabs are open in Firefox because the GC pause time increases dramatically. We reduce the GC pause time by 80% for such an environment. With our new approach, even users with 200+ open tabs get the same performance as users with just one single open tab.

Our research has major improvements for performance and security. We explain some security aspects of our approach but the main focus of this paper is performance.

2. Compartments

In this section we introduce compartments representing sub-heaps in our JavaScript VM. The concept of separating data using heuristic has a long history in computer science. Applying this concept to a VM architecture for JavaScript that is embedded in a browser still raises some challenging research questions.

The JavaScript programming language is widely used for web programming. It allows web developers to extend web sites with client-side executable code. JavaScript copies many names and naming conventions from Java, but the two languages are otherwise not closely related and have very different semantics. A lot of research was done in the area of memory management for Java but the results are often not applicable to JavaScript. First, there are fundamental differences between the two languages like dynamic typing and the dynamic behavior of JavaScript programs. Second, JavaScript programs written in web pages tend to have a very short execution time in comparison to Java applications. As with many dynamic languages, JavaScript objects are essentially associative arrays that lack static typing; object properties can be added and removed at runtime. JavaScript also provides a prototype-based inheritance mechanism to create complex object hierarchies.

For Firefox 4 we changed the way JavaScript objects are managed. Our JavaScript engine SpiderMonkey (sometimes also called TraceMonkey [2] and JägerMonkey, which are SpiderMonkey's trace-compilation and baseline just-in-time compilers) now supports multiple JavaScript heaps, which we also call compartments. All objects that belong to a certain origin (such as http://mail.google.com/ or http://www.bank.com/) are placed into a separate compartment as shown in Figure 2. The same-origin pol-

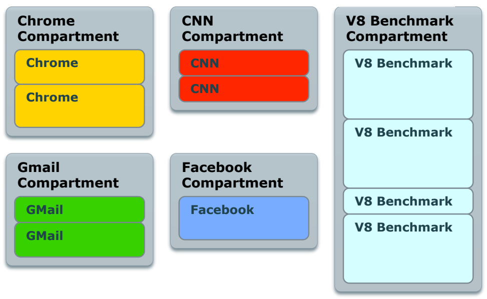

Figure 2. The new approach separates objects depending on their origin. New origins allocate a new compartment and just objects or the corresponding origin are placed in an arena.

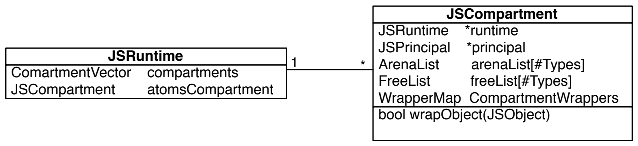
JSRuntime
ComartmentVector compartments
JSCompartment atomsCompartment
1
*
JSCompartment
JSRuntime *runtime
JSPrincipal *principal
ArenaList arenaList[#Types]
FreeList freeList[#Types]
WrapperMap CompartmentWrappers
bool wrapObject(JSObject)

Figure 3. The runtime holds all compartments. The compartments themselves hold their corresponding principal, a list of arenas where all objects and strings are allocated, all wrappers and provide functions to wrap objects and strings

icy (SOP) [19] is the central security policy in today's browsers. The policy defines that two documents from different origins cannot access each other's HTML documents using the DOM.

Our new compartment abstraction has a couple very important implications.

1. All objects created by a page from the same origin reside within the same compartment and hence are located in the same memory region. This improves cache utilization by reducing false sharing of cache lines. False sharing occurs when we are trying to operate on an object and we have to read an entire cache line of data into the CPU cache. In the old model JavaScript objects could be co-located with arbitrary other JavaScript objects from other origins. Such cross origin objects are used together very infrequently, which reduces the number of cache hits we get. In the new model most objects referenced by a website are tightly packed next to each other in memory, with no cross origin objects in between.

2. JavaScript objects (including JavaScript functions, which are objects as well) are only allowed to reference objects in the same compartment which means only same origin objects can reach each other. This invariant is very useful for security purposes. The JavaScript engine enforces this requirement at a very low level. It means that a google.com object can never accidentally leak into an untrusted website such as evil.com. Only a special object type can cross compartment boundaries. We call these objects wrappers. We track the creation of these cross compartment wrappers, and thus the JavaScript engine knows at all times what objects from a compartment are kept alive by outside references (through cross compartment wrappers). This allows us to garbage collect individual compartments, in addition to a global collection. We simply assume all objects referenced from outside the compartment to be live, and then walk the object graph inside the compartment. Objects that are found to be disconnected from the graph are discarded. With this new per-compartment GC we shortcut having to walk unrelated heap areas of a window (or tab) that triggered a GC.

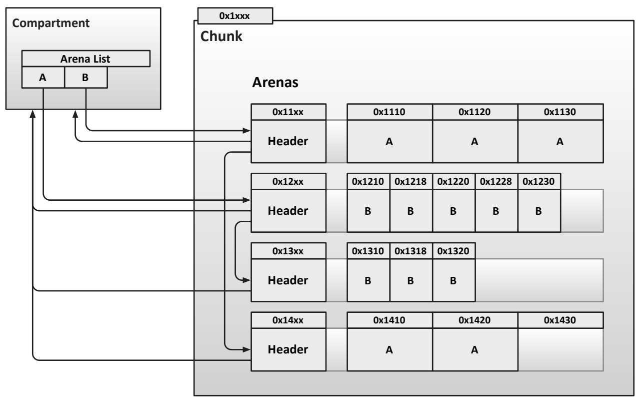

Figure 4. The basic data structures consists of 1MB chunks that are divided into 4KB arenas. Every arena has an arena header that stores basic information about the arena. The arena header also holds a reference to the corresponding compartment.

Our design is based on an allocation model introduced by Hanson [4]. A simplified example of our memory layout is shown in Figure 4. We allocate 1MB chunks from the operating system and split them up into 4KB arenas. Every arena has a header with basic information about the arena. It also holds a reference to the compartment the arena belongs to. With simple bit arithmetic (zeroing the last bits of each object address) we can get the address of the corresponding arena header. The arena header itself has a reference to the compartment it belongs to. So it is very easy and fast to get the corresponding compartment for each object. Each arena holds a certain type like strings or objects. This implies that all objects within an arena have the same size. A freelist keeps track of all free objects within the arena. The compartment holds a reference to the first arena header for a certain size class and this arena header holds the reference to the next arena with the same size class for the same compartment. This forms a linked list of arenas which all belong to the same compartment and hold the same types.

The compartments live in our runtime. Compartments are created for new origins and are destroyed whenever all objects contained within become unreachable. Each compartment maintains a list of arenas for each type of object that is allocated within. It also holds a freelist array with references to the next available allocation slots in an arena, or null if there are no slots available and a new arena has to be allocated. The wrapperMap of the compartment holds all wrapper objects that intercept cross-compartment communications. A detailed information about the wrappers is given in Section 3.

An interesting implication of the design of Firefox is that some of its code is written in JavaScript as well. This means that in the previous implementation, the internal code that is also called chrome code shares the same heap as all the external client side scripts. The new design treats this internal code as just another origin and all the objects are allocated in the chrome compartment.

2.1 Allocation

Allocating an arena from a chunk means now that no other compartment can allocate objects in there. In the previous model, threads were allocating multiple arenas from the arena list and kept them in the local thread storage. The allocation path had to be locked because other threads were also allocating arenas from the same list. With the new model we can get rid of almost all the locking. Once arenas are allocated they stay with the same compartment until they are empty and released.

Another popular optimization technique for JavaScript VMs is to create strings that are unique and immutable. We call them atoms. They are shared between the different scripts and no other string can have the same content as the actual atomized string. The main advantage comes from string comparison where the actual content comparison can be avoided. This "sharing of strings" might become a problem since we want to have as little cross-origin references as possible. Our solution for this issue is a separate compartment for all the immutable strings. Atomized strings also do not depend on any other strings. This implies that there are no references from the atoms compartment to other compartments. Allocating atomized strings is the only place where we need locking because different threads can allocate atoms at the same time. The function that creates atomized strings locks the allocation path and makes sure that only one thread is currently allocating from the atoms compartment.

3. Wrappers

As mentioned before, we want to minimize the cross-compartment references. But if they become necessary, we do not allow direct communication between these two objects from separate compartments. We delegate the communication technique to a wrapper object that is explained in this section. In JavaScript we distinguish between strings and objects. Strings and objects are both heap allocated but strings cannot have cyclic dependencies.

References between objects have to follow several rules now. As shown in Figure 5, an object o1 can only reference o2 if:

1. o1 and o2 reside in the same compartment and therefore have the same origin.
2. o2 is allocated from the atoms compartment meaning o2 is an immutable string.
3. o1 and o2 are in a different compartment and the VM explicitly allows this communication by adding a wrapper object representing o2 to the wrapper map in the compartment of o1.

$$o1 \rightarrow o2 \Rightarrow \left(\begin{array}{c} c(o1) == c(o2) \\ c(o2) == AtomsCompartment \\ (o1, o2) \in WrapperMap \end{array} \right)$$

Figure 5. An object cannot point to an arbitrary object in the JS heap any more.

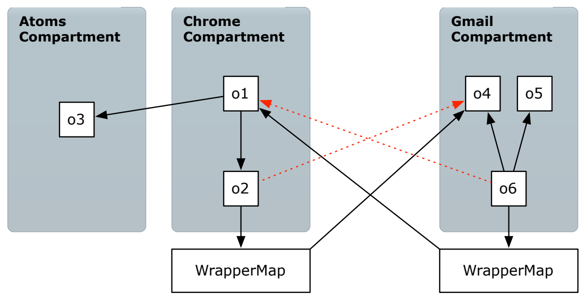

Figure 6. An overview of possible references between compartments.The red arrows represent the old way of communicating between two objects. In the new approach we add a wrapper objects between 2 objects that reside in different compartments.

Figure 6 shows all possible cross compartment communication mechanisms. The red slashed line represents the connection between two objects if they are in separate compartments. In the new model, each cross compartment reference is intercepted by a wrapper object that is stored in the wrapper map in each compartment.

References to an atom and therefore into the atoms compartment do not need a wrapper object.

Wrappers are not a new concept in Firefox, or browsers in general. In the past they were used to regulate how windows (or tabs) pass objects to each other. When a window or iframe tried to reference an object that belongs to a different window, we handed it a wrapper object instead. That wrapper object dynamically checks at access time whether the accessor window (also called the subject) is permitted to access the target object. If one Google Mail window is trying to access another Google Mail window, the access is permitted, because these two windows (or iframes) are same origin and hence it's safe to permit this access. If an untrusted website obtains a reference to a Google Mail DOM element, we hand it the same wrapper, and if it ever tries to access the Google Mail DOM Element the wrapper will, at access time, deny the property access because the untrusted website evil.com is cross origin with google.com.

A disadvantage of the Firefox 3.6 wrapper approach (which is similar to the way other browsers utilize wrappers) was the fact that these wrappers had to be injected manually at the right places in the C++ code of the browser implementation, and each wrapper had to do a dynamic security check at access time. With compartments we can do a lot better:

1. Since all objects belonging to the same origin are within the same compartment, and no object from a different origin is in that compartment, we can let all objects within a compartment reference other objects in the same compartment without a wrapper in between. Keep in mind that this does not just apply to windows but also to iframes. A single Google Mail session often uses dozens of iframes that all heavily exchange objects with each other. In the past we had to inject wrappers in between that kept performing dynamic security checks. This is no longer necessary, and there is an observable speedup when using iframe heavy web applications such as Google Mail.
2. Since all cross origin objects are in a different compartment, any cross origin access that needs to perform a security check can only happen through a cross compartment wrapper. Such a cross compartment wrapper always lives in a source compartment, and accesses a single destination object. When we create a cross compartment wrapper, we consult with the wrapper factory to see what kind of security policy should be applied. When evil.com obtains a reference to a google.com object, for example, we have to create a wrapper to that object in the evil.com compartment. When that wrapper is created, the wrapper factory applies a stringent cross origin security policy, which makes it impossible for evil.com to glean information from the google.com window. In contrast to our old wrappers, this security policy is static. Since only evil.com objects ever see this wrapper, and it only points to one single DOM element in the destination compartment, the policy does not have to be re-checked at access time. Instead, every time evil.com attempts to read information from the DOM element, the access is denied without even comparing the two origins.

3.1 Brain Transplants

A particularly interesting oddity of the JavaScript DOM representation is the existence of two objects for each DOM window (or tab or iframe), the inner window and the outer window. This split was implemented by web browsers a few years ago to securely deal with windows being navigated to a new URL. When such a navigation occurs, the inner window object inside the outer window is replaced with a new object, whereas the actual reference to window (which is the outer window) remains unchanged. If such a navigation takes the window to a new origin, we allocate the inner window in the appropriate new compartment. This of course creates now the problem that the outer window can possibly no longer directly point to the new inner window, because it is in a different compartment.

We solve this problem through brain transplants. Whenever an outer window navigates, we copy it into the new destination compartment. The object in the old compartment is transformed into a cross compartment wrapper that points to the newly created object in the destination compartment.

4. Partial GC

Having all JavaScript objects in the browser congregate in a single heap is suboptimal for a number of reasons. If a user has multiple windows (or tabs) open, and one of these windows (or tabs) created a lot of objects, it is likely that many of these objects are no longer reachable (garbage). When the browser detects such a state, it initiates a GC. Unfortunately though, since objects from different windows (or tabs) are intermixed on the heap, the browser has to walk the entire heap. If a number of idle windows are open, this can be quite wasteful, since those windows havent really created any garbage, so whenever a window with heavy activity triggers a GC, much of the GC time is spent walking unrelated parts of the global object graph.

In Firefox this problem is even more pronounced than in other browsers, because our UI code (also called chrome code, not to be confused with Google Chrome) is implemented in JavaScript, and there are a lot of chrome (UI) objects alive at any given moment. These UI objects tend to stick around and every time a web content window causes a GC, Firefox spends a lot of time figuring out whether chrome objects are still alive instead of being able to focus on the active web content window.

The new approach allows us to perform partial-GC on single compartments. A single compartment GC or per-compartment GC is triggered whenever the allocation of a single compartment reaches some watermark that is set after a GC depending on the working set size. As a simple example, assume that a single compartment GC is triggered when 10MB of JavaScript objects are allocated. If we reach this level, we also check the overall allocation of all compartments. If the overall allocation exceeds 150% of the triggering compartment allocation (or 15MB in this example) we perform a global GC. There exist other GC triggers in the browser but they are not relevant to the per-compartment GC approach and beyond the scope of this paper. We also can not get rid of the global GC because the new approach introduces the possibility of cyclic data structures between compartments. Two objects in separate compartments that point to each other would never get collected with only per-compartment GCs since the wrapperMaps keep them alive.

4.1 Marking

In order to find all reachable objects for a global GC we traverse the object graphs beginning with following roots: First, we perform a conservative stack scan and mark all objects that are reachable from the native C stack. Then we mark all explicit roots that are stored in a roots hashtable followed by marking all global objects.

Marking reachable objects for a single-compartment GC follows the same scheme as the marking for the global GC with one additional step. As mentioned before we assume all objects in other compartments to be alive. Since there are no direct pointers between compartments, marking all wrapper references from other compartments is sufficient to capture all reachable objects. The marking function checks every reference if the corresponding object is in the currently in the GC involved compartments. Getting the compartment identity is done with simple pointer arithmetic and is very cheap as described in Figure 4.

4.2 Sweeping

The sweeping phase for a global GC consists of traversing each arena and checking for unreachable (unmarked) objects. The advantage for a single compartment GC is that we do not have to traverse all arenas. It is sufficient to traverse only arenas that are allocated from the compartment involved in the single compartment GC since all other objects are considered alive as mentioned before. The sweeping process touches each object and checks the mark bit. If the mark bit is not set, a finalizer is called for the object and the location is put on the freelist of the arena.

5. Granularity

Finding the right granularity for compartmentalizing web-content is the key for success. On the one hand we have the old approach with a single JavaScript heap and all objects regardless of their origin are intermixed on the heap. The other side of the spectrum is not that easy to define. "Web programs are easy to understand intuitively but difficult to define precisely." [17].

A web application like GMail consists of many sub-structures. Typical components are parent-pages containing images, script-libraries, embedded frames, popup pages for chatting and messages. Placing each of these items into separate compartments would result in many compartments just for a single page like GMail. In order to argue that one compartment per origin is the right choice, we can compare it with an implementation where we separate objects based on iframes. The HTML `<iframe>` tag defines an inline frame that contains another document and is supported by all major browser vendors. There is no general way of telling how many iframes a web page has but in order to compare our approach with a solution where each iframe gets its own compartment we compare typical web pages in Table 1. We compare our approach with an implementation that creates a new compartment for each iframe in Table 2. We can see that the finer granularity would work for some pages like Ebay and Digg, but for other pages the number of compartments increases dramatically. Techcrunch for example would have 154 compartments instead of 11. For GMail, the number of wrappers would increase from 183 to 5654.

Alias	URL
280s	280slides.com
AMAZ	amazon.com
BING	bing.com
BLOG	blogger.com
DIGG	digg.com
EBAY	ebay.com
FBOK	facebook.com
FLKR	flickr.com
GDOC	docs.google.com
GMAP	maps.google.com
GMIL	gmail.com
GOGL	google.com
HULU	hulu.com
ISHK	imageshack.us
TECH	techcrunch.com
V8BE	V8.googlecode.com/svn/data/benchmarks/v6
YTUB	youtube.com

Table 1. Selected JavaScript-enabled web sites.

Alias	Origin	Wrappers	IFrame	Wrappers
280s	1	26	2	85
AMAZ	4	280	16	563
BING	1	80	3	105
DIGG	3	114	3	115
EBAY	1	48	1	50
FBOK	1	249	6	445
FLIKR	3	185	23	1094
GDOC	6	552	7	277
GMAP	1	88	2	82
GMIL	2	183	9	5654
GOGL	1	60	2	209
HULU	1	103	10	245
ISHK	6	776	41	1396
TECH	11	2324	154	3094
V8BE	1	35	1	35
YTUB	2	183	7	204

Table 2. Compartments and corresponding cross compartment pointers if we create new compartments per origins or iframes.

6. Processes

Another question is how compartments compare to per-tab processes as they are used by Google Chrome and Internet Explorer. Both processes and compartments shield JavaScript objects against each other. The most important distinction is that processes offer a stronger separation enforced by the processor hardware, while compartments offer a pure software guarantee. However, on the upside compartments allow much more efficient cross compartment communication that processes code.

With compartments, cross origin websites can still communicate with each other with a small overhead (governed by certain cross origin access policy), while with processes cross-process JavaScript object access is either impossible or extremely expensive. In future browsers we will likely see both forms of separation being applied. Two web sites that never have to talk to each other can live in separate processes, while cross origin websites that do want to communicate can use compartments to enhance security and performance.

The space overhead can be shown by simply opening an empty tab and measuring the increased memory consumption. Opening another tab in Chrome creates a new process with about 30MB. Open another tab in Firefox is about 2.2MB and Safari about 10MB.

Another drawback that is introduced by the process level separation comes from the object communication mechanism. Two objects that want to communicate with each other have to go through an expensive inter process communication mechanism. A message sent from an object A to another object B does not have any guarantee to be received from B if there is no synchronization in place. The run-to-completion semantics defines that a state-machine has to complete processing one event before it can start processing the next.

Google Chrome supports 4 different process models: 1) monolithic process, 2) process per browsing instance, 3) process per site instance and 4) process per site. Models 1) and 2) do not provide memory protection across multiple origins. Model 3), which is enabled by default, and model 4) still do not prevent origins that are embedded with the iframe tag from accessing objects from the parent page because they are all execute in one process.

7. Evaluation

To evaluate our compartmental memory management approach, we implemented it in the open source JavaScript VM SpiderMonkey [15], which is used by Mozilla Firefox. As a result of this choice we are able to provide benchmark numbers for in-browser synthetic benchmarks as well as actual JavaScript web applications.

All experiments were performed on a Mac Pro with 2 x 2.66 GHz Dual-Core Intel Xeon processor and 4 GB RAM running MacOS 10.6 and beta version 10 of Firefox 4.0 that uses the compartments mechanisms we have introduced in this paper as its default configuration. It is very easy to rerun the benchmarks by just setting the `javascript.options.mem.gc_per_compartment` option in the `about:config` page of Firefox. This section uses baseline implementation or just base where we only perform global GCs and per-compartment implementation or comp where we also perform per-compartment GCs.

7.1 Cost in Space

The first question to answer is whether the new approach improves the memory footprint of the VM or even introduces some space overhead. There are two scenarios that influence the space-overhead in a positive and negative way. Since we do not intermix objects of different origins within arenas any more, we always have to allocate a new arena if all arenas are full for a certain compartment. So we end up allocating arenas even if there are some empty slots in arenas that belong to another compartments. On the other hand, if there are reachable objects within an arena, we cannot return the arena to the OS.

With the new approach it is more likely that objects with the same lifetime end up in the same compartment. Whenever we close a tab, the corresponding compartments including all its arenas are very likely to become garbage. Once there are no reachable objects within the arenas, we can return them to the OS. There are no reachable objects from other domains that remain in the arena.

Figure 7 shows the difference between the old model and the new model. In this experiment, we open 50 tabs with popular web pages and close one after another. The y-axis represents the number of allocated 4KB arenas. As expected, the new approach has a higher peak demand because allocated arenas belong to only one origin. The difference for 50 tabs is for this example around 13% or 15MB. During the closing process, the new model shows its advantages. Since closing a tab releases all objects from a certain origin, the corresponding arenas become empty. The results of Figure 7 also show that our new approach is going towards a generational GC model. We can clearly see that objects separation based on their origin shows better results than just intermixing them with other objects. This aspect is an interesting outcome that will lead to further investigation.

One of the key factors for our partial GC approach is the volume of missed space that does not get freed because we assume all objects are reachable within this space. We changed the way our per-compartment GC works in order to get detailed information about unreclaimed objects because of our partial GC approach. Table 3 shows detailed numbers for the GC workloads. We open 50 tabs in the browser with popular websites and once all of them are fully loaded we start the V8 benchmark suite. Whenever we would trigger a per-compartment GC, we perform a full GC but do not reclaim objects that are not part of the compartment that triggered the GC. *Base Reachable* means all objects that are reachable in the JavaScript VM. *Comp Reachable* represents all objects that are reachable within the compartment that triggered the per-compartment GC. Comp Reachable is 0 if a global GC is performed because there is no special compartment involved in the GC. *Finalized* represent all objects that are finalized during the GC event. *Missed* represents the number of unreachable objects that

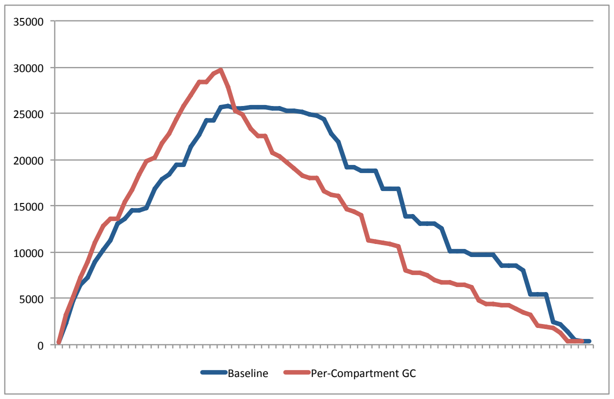

Figure 7. Opening 50 tabs and close them again with the baseline and per-compartment approach. We can see a higher memory consumption peak for the opening process with the new approach but once we close tabs, we can also deallocate arenas faster.

are not reclaimed because another compartment performed per-compartment GC. *GC Type* is either global GC or per-compartment GC where a GC is performed for a single compartment. Relative values are calculated as follows:

$$Reachable\,Rel. = \frac{Comp\ Reachable}{Base\ Reachable} * 100\%.$$

$$Missed\,Rel. = \frac{Missed}{Finalized} * 100\%.$$

$$Relative\,to\,Total = \frac{Missed}{Missed{+}Finalized{+}Base\ Reachable} * 100\%$$

The first three global GCs happen during loading of the 50 tabs. Once we start the V8 benchmark suite we only see per-compartment GC until we shut down the browser. The shutdown process performs the last three global GCs. The numbers show that most of the time we only mark up to 7% of all reachable objects. Only during the splay benchmark where a huge splay tree is created and modified the actual reachable objects represent around 60% of the whole browser heap. More interesting is the ratio between finalized and missed objects. We can see that during the benchmark we create around 3% garbage in other compartments that is not reclaimed. During the splay benchmark we see a spike at 288%. This happens during the creation of the splay tree where a per-compartment GC is triggered but almost no objects are finalized because the whole tree is alive. Relative to Total measures the ratio between total heap space and missed to finalize. We can see that we only miss to reclaim about 2% of the heap space because of all the per-compartment GCs. We can also see that the 288% value during the splay benchmark has no impact on the overall ratio. Note that the number of GC events differs from the results in Table 4 because our instrumentation increased the GC pause time and therefore also influenced the benchmark scores.

7.2 V8 Benchmark

The V8 suite runs each benchmark for one second and computes a score per benchmark and an overall score based on each individual score. Since the benchmark runs for one second, the amount of memory that is used varies. An allocation-heavy benchmark allocates more objects and therefore more memory in the same amount of time if the allocation becomes faster and GC pause time is reduced. We also performed VM internal measurements in order to discuss the GC events happening during running the V8 benchmarks in more detail. We use the time stamp counter `rdtsc` [8] in order to measure the duration of each GC event.

Base Reachable	Comp Reachable	Reachable Rel.	Finalized	Missed	Missed Rel.	Relative to Total	GC Type
239446	0	0.00%	412554	0	0.00%	0.00%	Glob
407492	0	0.00%	727585	0	0.00%	0.00%	Glob
837631	0	0.00%	926075	0	0.00%	0.00%	Glob
837497	4330	0.52%	826047	123	0.01%	0.01%	Comp
837259	4092	0.49%	826310	123	0.01%	0.01%	Comp
837377	4141	0.49%	1299044	24745	1.90%	1.14%	Comp
837347	4111	0.49%	870836	24745	2.84%	1.43%	Comp
837345	4109	0.49%	870880	24745	2.84%	1.43%	Comp
837344	4108	0.49%	870846	24745	2.84%	1.43%	Comp
837352	4116	0.49%	847193	24745	2.92%	1.45%	Comp
837348	4112	0.49%	870836	24745	2.84%	1.43%	Comp
837891	4576	0.55%	766282	28833	3.76%	1.77%	Comp
839837	6522	0.78%	762931	28833	3.78%	1.77%	Comp
841563	8248	0.98%	761453	28833	3.79%	1.77%	Comp
838833	5518	0.66%	764411	28833	3.77%	1.77%	Comp
840674	7359	0.88%	762224	28833	3.78%	1.77%	Comp
871243	37846	4.34%	836353	31299	3.74%	1.80%	Comp
887190	53793	6.06%	821184	31299	3.81%	1.80%	Comp
894486	61089	6.83%	813888	31299	3.85%	1.80%	Comp
904673	71276	7.88%	803667	31299	3.89%	1.80%	Comp
846209	12812	1.51%	862147	31299	3.63%	1.80%	Comp
875113	41716	4.77%	833224	31299	3.76%	1.80%	Comp
874065	40668	4.65%	755349	31299	4.14%	1.88%	Comp
888829	55432	6.24%	819508	31299	3.82%	1.80%	Comp
897413	64016	7.13%	810924	31299	3.86%	1.80%	Comp
839779	6382	0.76%	868580	31299	3.60%	1.80%	Comp
872909	39512	4.53%	835428	31299	3.75%	1.80%	Comp
887213	53816	6.07%	821124	31299	3.81%	1.80%	Comp
897191	63794	7.11%	811149	31299	3.86%	1.80%	Comp
839420	6023	0.72%	868966	31299	3.60%	1.80%	Comp
872702	39305	4.50%	835635	31299	3.75%	1.80%	Comp
856356	22985	2.68%	938002	35474	3.78%	1.94%	Comp
868587	35091	4.04%	1289506	37802	2.93%	1.72%	Comp
1924649	1091153	56.69%	13118	37802	288.17%	1.91%	Comp
2127944	1294473	60.83%	2016589	38177	1.89%	0.91%	Comp
2127922	1294451	60.83%	3507997	38177	1.09%	0.67%	Comp
2127961	1294490	60.83%	4648974	38177	0.82%	0.56%	Comp
378561	0	0.00%	2129060	0	0.00%	0.00%	Glob
43255	0	0.00%	335505	0	0.00%	0.00%	Glob
31497	0	0.00%	11758	0	0.00%	0.00%	Glob

Table 3. GC workloads for 50 open tabs and running the V8 Benchmark.

	1 Base	1 Comp	50 Base	50 Comp
Richards	7929	7932	8211	8084
DeltaBlue	4198	5263	2142	4985
Crypto	8634	8598	8779	8596
RayTrace	3510	3527	1698	3464
EarleyBoyer	4357	4550	1514	3807
RegExp	1711	1692	1624	1651
Splay	5012	5134	3529	5041
Score	4505	4692	3017	4511

Table 4. Results of the V8 benchmark suite (higher is better). The numbers represent running the benchmark suite in a single tab for the baseline and per-compartment approach (Comp) and opening 50 typical web pages and running the benchmark suite for the baseline and per-compartment GC approach.

Table 5 shows the results for running the V8 benchmarks for the baseline and our new approach. We can see that the reduced workload due to the partial GC increased the number of performed GCs from 63 to 75. The total time we spend in marking increases because we are performing more GCs but the average time we spend in marking reduces around 12%. The increase in finalization time comes from the fact that more objects have to be finalized. As explained in Section 4, we have to check the mark bit of every single object during sweeping. Since we encounter less marked objects and more unreachable objects, the time we spend in finalization increases. Marking less objects and finalizing more objects indicates a good separation technique.

Figure 8 through Figure 11 show the mark - sweep ratio for each GC event for the V8 benchmarks. Figure 8 shows the marking and sweeping ratio for starting the browser, running the V8 benchmarks and closing the browser again with our baseline approach. Figure 9 shows the marking and sweeping ratio with our new per-compartment GC model. We can see that even for a sin-

	Base	Average	Comp	Average	Relative
GC Events	63	-	75	-	+16%
Marking	2891	46	3075	41	-12%
Sweeping	2693	43	3319	44	+3.4%
Total	6117	97	6583	88	-11%

Table 5. Basic internal measurements for the V8 benchmark. The numbers represent 1E6 cycles measured with `rdtsc`.

gle tab we reduce the time spent in marking because we only perform the GC in the benchmark compartment and do not include the browser internal chrome compartment. Table 4 shows that the benchmark score increases from 4505 to 4692 for a single open tab. The big spike at the end is caused by the allocation intensive splay benchmark. The finalization spike at the very end is caused by the shutdown of the browser.

The real strength of the new approach comes with many open tabs. Figure 10 and Figure 11 show the mark-sweep ratio with 50 other open tabs. We start the browser, open 50 tabs, wait until they are fully loaded and start the V8 benchmark in a new tab. We can see that marking time dominates the GC pause time in Figure 10. If we compare it to Figure 11 we can clearly see the improvements. We perform global GCs at the beginning because we open many web pages and the overall memory footprint increases. Once we start the V8 benchmark we see that the per-compartment GC is triggered because only the benchmark origin creates objects. There is one spike in the middle of the benchmark where the browser decides to perform a global GC. This is either caused by internal timers of the browser or an overall increase of the memory footprint. Note that this doesn't happen all the time as can be seen in Table 3. We can see that the time is identical to the baseline approach for this single spike. Table 4 shows that the benchmark score increases from 3017 to 4511.

Running the benchmark with an additional 50 open tabs reaches now the same performance as running the benchmark in a single tab without our new model. The average number of cycles for each GC event reduces from 885*10E6 to 294*10E6. If we only consider the interval where the benchmark is running and exclude the start and shutdown overhead we reduce the average numbers of cycles from 998E6 to 170E6 which is a reduction of 83%.

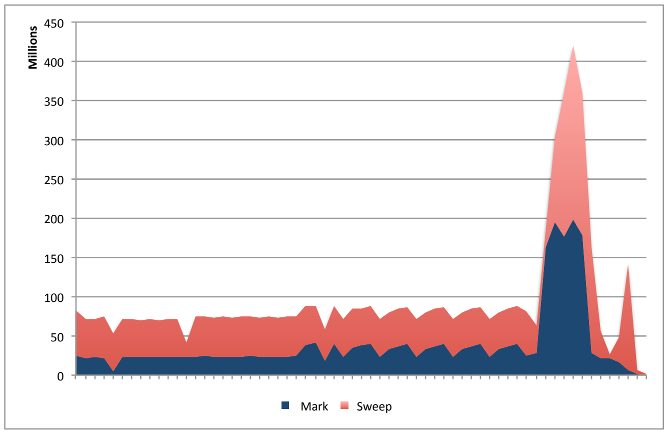

Figure 8. Running the V8 Benchmark Suite with a single tab with baseline approach. The y-axis shows a stacked representation of cycles measured with `rtdsc`.

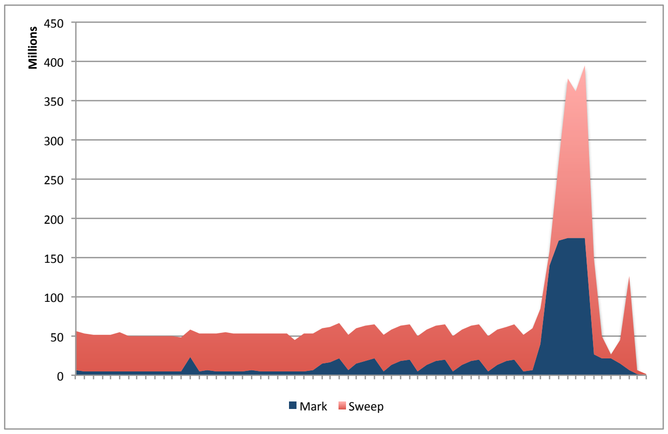

Figure 9. Running the V8 Benchmark Suite with a single tab with per-compartment GC.

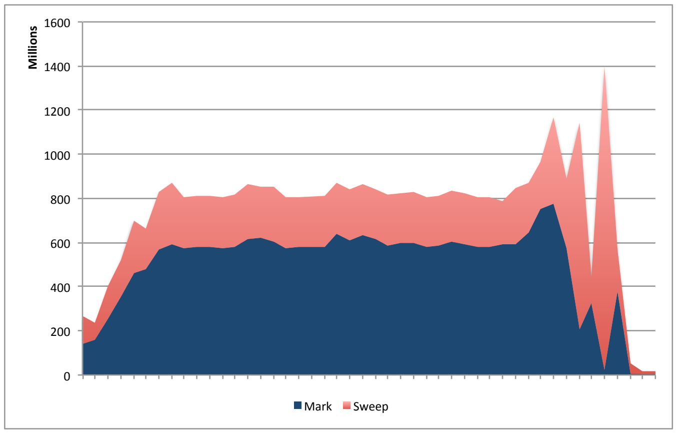

Figure 10. Opening 50 tabs with popular web pages and running the V8 Benchmark Suite with baseline approach.

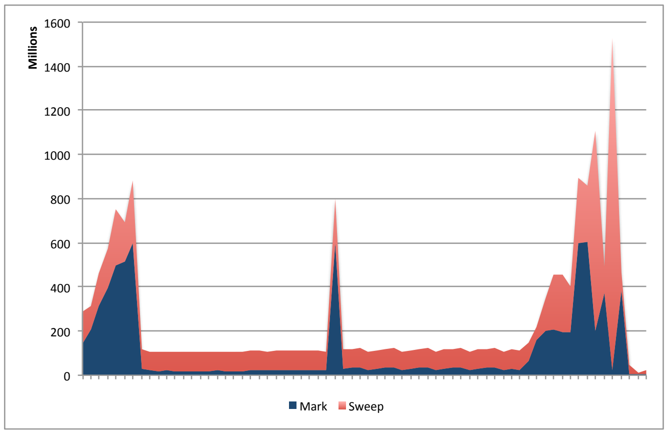

Figure 11. Opening 50 tabs with popular web pages and running the V8 Benchmark Suite with the new per-compartment GC.

7.3 Kraken Benchmark Suite.

Table 6 shows the kraken benchmark [14] results. The benchmark was running in a browser with websites loaded from Table 1 except the V8 benchmark suite. We can see an overall performance increase from 6.9% due to shorter GC pause times. The *From* column represents the baseline and the *To* column represents the per-compartment GC approach. The new approach also introduces more stability for the individual benchmarks. As can be seen in Table 6, the random noisy for the individual benchmarks is reduced.

Benchmark	Base [ms]	+/- [%]	Comp [ms]	+/- [%]
astar	1236.3	5.0	1182.7	5.8
beat-detection	457.0	12.9	418.5	3.4
dft	496.8	13.2	473.5	3.6
fft	343.2	13.5	348.6	3.7
oscillator	290.8	0.7	290.8	0.7
gaussian-blur	492.5	0.2	492.0	0.2
darkroom	221.7	0.5	221.0	0.2
desaturate	487.6	5.1	477.1	0.2
parse-financial	131.3	32.7	111.8	1.3
stringify-tb	96.2	51.2	71.8	2.3
aes	231.6	23.4	234.8	9.4
ccm	154.5	2.1	161.3	8.2
pbkdf2	313.4	23.8	237.6	7.4
sha256-it	198.2	39.0	95.9	2.7
TOTAL	5151.1	1.8	4817	1.7

Table 6. Kraken benchmarks

7.4 SunSpider Benchmarks

One of our claims was that we improve locality of reference with our new approach. Since we do not allocate objects in already used arenas from another compartment and rather allocate a new arena, we place objects right next to other objects from the same origin. Running the SunSpider benchmark suite is an indicator for a better locality during the benchmark run because there is no GC event during the benchmark. Also the locking that is removed for arena allocation increases performance. The benchmark suite executes all benchmarks 10 times with a forced GC in between that does not impact the benchmark scores. SunSpider is a time based benchmark suite where actual execution time is measured. Table 7 shows the results of the SunSpider benchmarks. We can see a 3% improvement with the new allocation scheme.

7.5 Non Benchmarks

Reducing the GC pause time has also other advantages than increasing benchmark scores. An everyday Firefox user cares more about the performance for real workloads. Our new approach greatly improves the performance of all allocation heavy web apps such as JavaScript based animations and games. The GC pause time during an animation is no longer related to the number of open tabs and users do not have to close all other tabs in order to get the best performance for JavaScript based games.

8. Related Work

Jones and Lins [9] describe basic GC algorithms that are also used in our implementation. The current implementation of the memory management system in SpiderMonkey is based on the research from Hanson [4].

Optimizing allocation patterns to improve the locality of reference in the virtual memory [16] and cache [11] has been studied over many years. Basic implementation like the "first-fit" approach [10] or improvements like the "better-fit" approach [22] still show bad reference locality characteristics. We use object separation based on their origin to get better reference locality. For example our internal objects created from our chrome code do not share pages with objects allocated from web sites any more.

Benchmark	Base [ms]	Comp [ms]	Relative [%]
cube:	16.1	15.7	2.48
morph:	16.1	15.8	1.86
raytrace:	36.5	36.2	0.82
binary-trees:	19.9	19.1	4.02
fannkuch:	13	12.9	0.77
nbody:	4	4	0.00
nsieve:	5	5	0.00
3bit-bits-in-byte:	0.5	0.5	0.00
bits-in-byte:	6.7	6.7	0.00
bitwise-and:	1.3	1.2	7.69
nsieve-bits:	4.3	4.2	2.33
recursive:	21.3	20.9	1.88
aes:	10.5	10.3	1.90
md5:	5.3	5.2	1.89
sha1:	2.6	2.6	0.00
format-tofte:	20.1	19.4	3.48
format-xparb:	13.4	12.8	4.48
cordic:	8.3	4.6	44.58
partial-sums:	7.9	7.8	1.27
spectral-norm:	3.2	3.2	0.00
dna:	11.8	11.9	-0.85
base64:	3.3	3.1	6.06
fasta:	12.4	12.7	-2.42
tagcloud:	22.4	21.4	4.46
unpack-code:	29.6	28.9	2.36
validate-input:	5.3	4.9	7.55
TOTAL	300.7	291.1	3.19

Table 7. SunSpider benchmark suite.

Reis et al. [17] show the various process models supported by Google Chrome. They compare different process isolation models (monolithic process, process-per-browsing-instance, process-per-site and process-per-site-instance) that are all supported by Google Chrome. In contrast to our work, they attempt to create new processes for new domains. A more detailed discussion about the differences can be found in Section 6.

Microsoft [12] also uses OS processes to isolate tabs from one another in Internet Explorer 8. This protection mechanism is insufficient from a security standpoint since a user may browse multiple mutually distrusting sites in a single tab via iframes.

In more recent work from Microsoft Research, Wang et al. [23] present a secure web browser constructed as a multi-principal OS. The browser is called Gazelle and its kernel is an operating system that exclusively manages resource protection and sharing across web site principals. The main drawback is the performance. The page load time for a site like nytimes.com increases to around 6 seconds.

Hirzel et al. [6] do an interesting analysis on the connectivity of heap objects. They show the importance of understanding the connectivity of the heap objects and give hints on improving existing partition models. Their research is focused on Java but the overall connectivity idea is also relevant for JavaScript. Hirzel [5] also shows in his PhD thesis a connectivity based GC approach that relies on object connectivity analysis. Similar to our approach they try to place objects with the same lifetime and access frequency in the same memory area.

Seidl et al. [20] present a profile-driven object lifetime and access frequency predictor. They reduce the number of page faults by placing highly referenced objects next to each other on a small set of pages. Short lived objects on the other hand, are placed on a small set of different pages.

Cox et al. [1] use multiple VMs to completely isolate web applications. They present a solution to prevent cross origin communication with an overhead of up to 9 seconds to start a new browsing instance.

Gries et al. [3] present the OP web browser which is based on a browser- level information-flow tracking system. It enables them to analyze browser-based attacks after they have happened and show the possible root of the attack.

More recently, Inoue et al. [7] made a study of memory management for web-based applications on multicore processors. They compare a traditional and a region-based memory allocator for PHP applications and show speedups of up to 27%. They introduce a freeAll function that can be called from an application once all of the objects on the heap can be deallocated.

Richards et al. [18] present a study of currently used JavaScript benchmarks. They compare the behavior of V8 and SunsSpider benchmarks with popular web pages. One of the outcomes of this research is that the overall lifetime of benchmark objects is not comparable to actual web pages.

9. Conclusions

We demonstrated the advantages and an efficient implementation of per-compartment GC. We add another layer of abstraction to the JavaScript heap and separate JavaScript data based on their origin. Partial GC on a single compartment reduces the workload for the GC and therefor reduces the GC pause time. Our experiments show that the GC pause time for running the V8 benchmarks with 50 other open tabs is reduced by up to 83%.

The foundation we laid with the compartments work will also enable a number of future extensions. Since we now cleanly separate objects belonging to different tabs, future changes to our JavaScript engine will permit us to not only perform JavaScript GC for individual compartments, but we will also be able to do so in the background on a different thread for tabs with inactive content.

Acknowledgments

We want to thank Jason Orendorff and Blake Kaplan from Mozilla that worked very hard in order to make this research happening. Michael Bebenita and Mason Chang from UC Irvine gave very valuable feedback. Other important help came from the Mozilla community. They tested our approach, reported bugs and helped us fixing them.

References

[1] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 350–364. IEEE Computer Society, 2006. doi: 10.1109/SP.2006.4.

[2] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478. ACM Press, 2009. doi: 10.1145/1542476.1542528.

[3] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 402–416. IEEE Computer Society, 2008. doi: 10.1109/SP.2008.19.

[4] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software - Practice and Experience*, 20:5–12, 1990. doi: 10.1002/spe.4380200104.

[5] M. Hirzel. *Connectivity-Based Garbage Collection*. PhD thesis, Department of Computer Science, University of Colorado at Boulder, 2004.

[6] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Proceedings of the International Symposium on Memory Management*, pages 36–49. ACM Press, 2002. doi: 10.1145/512429.512435.

[7] H. Inoue, H. Komatsu, and T. Nakatani. A study of memory management for web-based applications on multicore processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 386–396. ACM Press, 2009. doi: 10.1145/1542476.1542520.

[8] Intel. Using the RDTSC instruction for performance monitoring, 1997.

[9] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., 1996.

[10] D. E. Knuth. *Fundamental Algorithms, The Art of Computer Programming, chapter 2*, volume 1. Addison Wesley, 2nd edition, 1973.

[11] S. McFarling. Program optimization for instruction caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM Press, 1989. doi: 10.1145/70082.68200.

[12] Microsoft. What's new in Internet Explorer 8, 2008. URL http://msdn.microsoft.com/en-us/library/cc288472.aspx.

[13] Mozilla. Firefox web browser and Thunderbird email client, 2011. URL http://www.mozilla.com.

[14] Mozilla. Kraken JavaScript benchmark, 2011. URL http://krakenbenchmark.mozilla.org/.

[15] Mozilla. SpiderMonkey (JavaScript-C) engine, 2011. URL http://www.mozilla.org/js/spidermonkey/.

[16] J. Peachey, R. Bunt, and C. Colbourn. Some empirical observations on program behavior with applications to program restructuring. *IEEE Transactions on Software Engineering*, 11:188–193, 1985. doi: 10.1109/TSE.1985.232193.

[17] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the European Conference on Computer Systems*, pages 219–232. ACM Press, 2009. doi: 10.1145/1519065.1519090.

[18] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2010. doi: 10.1145/1806596.1806598.

[19] J. Rudermann. The same origin policy, 2001. URL https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

[20] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23. ACM Press, 1998. doi: 10.1145/291069.291012.

[21] StatCounter. Global Stats, 2011. URL http://gs.statcounter.com/.

[22] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 30–32. ACM Press, 1983. doi: 10.1145/800217.806613.

[23] H. J. Wang, C. Grier, E. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the USENIX Security Symposium*, pages 417–432. USENIX, 2009.