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 substantially. Rather than sequentially navigating static web sites, modern web browsers often manage a large number of simultaneous 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.

Memory management and garbage collection (GC) are now a severe bottleneck within the browser and the JavaScript virtual machine (VM) executing client-side web programs. While previously browsing speed was mostly degraded by rendering and network latency, GC pause times have now become an important factor of browser performance.

Architectural changes such as multiple browser tabs have changed the way users browse the web. The underlying JavaScript execution model has not kept up with this evolution. The implementation of the memory management subsystem of JavaScript VMs do 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 JavaScript VMs. As a result, web pages loaded in separate tabs encounter interference with each other in ways that affect their memory management, security and performance.

Some web browsers such as Google Chrome or Microsoft Internet Explorer 8 address this separation problem by creating a new process for each new tab or origin. This is good for security since process boundaries act like "hardware fences" between browsing instances and memory management can be handled completely separately for every tab. Chrome also spawns separate instances of the JavaScript VM for every process. Since the created browsing instances are heavyweight, Chrome limits the number of processes to 20.

There are at least two problems with this approach: Certain web features such as iframe navigation require pages to maintain references to objects belonging to other pages. In order to support this pattern, Chrome loads such pages into the same rendering process, losing any benefits of process separation along the way. Furthermore, creating a new process for every origin is not an option for environments with limited resources such as mobile devices. One of the design constraints of our new system 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. Figure 1 depicts the JavaScript heap with some tabs open in the browser. In this example we open some tabs and load popular web pages. The objects are not separated on the heap and it is likely that all the objects from different origins interleave within the heap. A Facebook object might reside 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 illustrates 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 must be accessed.

Our research proposes a new layer of abstraction for the JavaScript heap. We split the JavaScript heap into sub-heaps, which

ISMM'11, June 4–5, 2011, San Jose, California, USA.

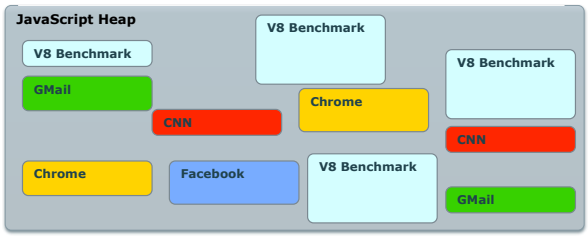

Figure 1. The previous implementation allows objects of different origins to be allocated in the same memory region.

we call compartments. JavaScript objects that are allocated from a 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 [16]. Firefox has about 400 million daily users with market share between 25% and 30% according to [24].

Having many open tabs is not unusual any more. User reports that are collected at Mozilla show that some users have 200+ open tabs. Running benchmarks in such an environment have shown drastic performance impacts. For example, the V8 benchmark score drops from 4511 to 3017 when 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 now 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 heuristics 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 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 such as dynamic typing and the dynamic behavior of JavaScript programs. Second, JavaScript programs written in web pages tend to have a 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 [3] 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

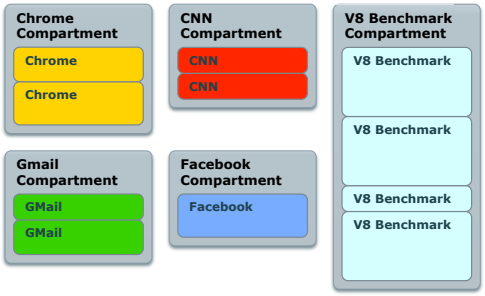

Figure 2. The new approach separates objects depending on their origin. New origins allocate a new compartment and only objects with the associated origin are placed in an arena.

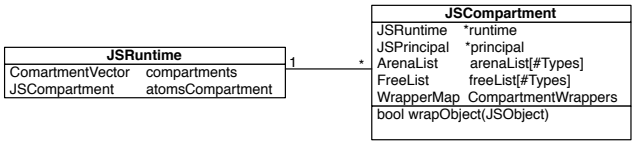

Figure 3. The runtime holds all compartments. The compartments themselves hold their corresponding principal, a list of arenas where all objects and strings are allocated, all wrappers and provide functions to wrap objects and strings

a separate compartment as shown in Figure 2. The same-origin policy [22] is the central security policy in today's browsers. The policy specifies that two documents from different origins cannot access each other's HTML documents using the DOM.

Our new compartment abstraction has a couple of 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 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 useful for security purposes. The JavaScript engine enforces this requirement at a 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.

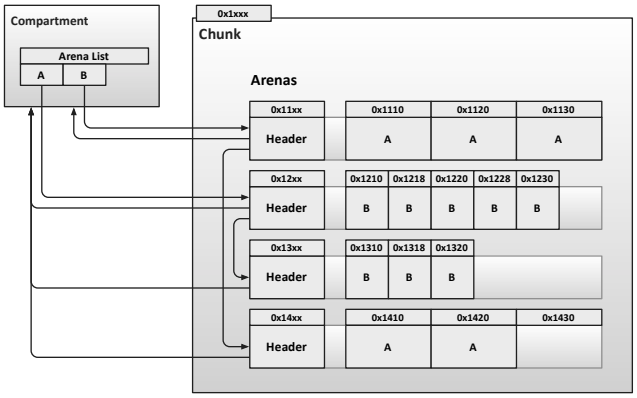

Figure 4. The basic data structures consists of 1MB chunks divided into 4KB arenas. Every arena has a header that stores basic information about the arena. The arena header also holds a reference to the corresponding compartment.

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 many 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 much time figuring out whether chrome objects are still alive instead of focusing on the active web content window.

Our design is based on an allocation model introduced by Hanson [5]. 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. With simple bit arithmetic (zeroing the last bits of each object address) we can obtain the address of the corresponding arena header. The arena header itself has a reference to the compartment it belongs to. This arrangement makes it easy and fast to lookup the corresponding compartment for each object.

Each arena holds a certain type such as strings or objects. This implies that all objects within an arena have the same size because all objects are allocated with an initial number of slots. If the objects grows beyond the initial size, additional memory has to be allocated for the object. Strings also have the same size and the actual payload is stored in dynamic memory. A free-list keeps track of all free objects within the arena and the reference to the first free object is stored in the arena header.

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. These links form a list of arenas which all belong to the same compartment and hold the same types. For fast allocation we have an array per compartment representing all size classes referencing the next available allocation slot in an arena or null if there are no slots available and a new arena must be allocated.

The compartments themselves live in our runtime. Compartments are created for new origins and are destroyed whenever all objects contained within become unreachable. The wrapperMap of the compartment holds all wrapper objects that intercept cross-compartment communications. The general Wrapper concept is explained in Section 3.

2.1 Allocation

Allocating an arena from a chunk now means that no other compartment can allocate objects in the same arena. In the previous model, threads allocated 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. After a GC, all arenas with available slots were placed on a global list and threads had to use a lock again for allocation. With the new model we can dispense with almost all the locking because arenas stay within the same compartment. After a GC we can simply traverse all arenas that already belong to a certain compartment without locking in order to allocate new objects. Once arenas are allocated they stay within the same compartment until they are empty and released.

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

Figure 5. An object can no longer point to an arbitrary object in the JS heap.

Another popular optimization technique for JavaScript VMs is to create strings that are unique and immutable. We call them atoms but the technique is also called "interning" in Java VMs. The strings 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 fine grained locking because different threads can allocate atoms at the same time. The function that creates atomized strings locks the allocation path and ensures that only one thread is currently allocating from the atoms compartment at a time.

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 must now follow several rules. 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.

Figure 6 shows all possible cross compartment communication mechanisms. The red dashed line represents the connection between two objects when 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 atom's 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

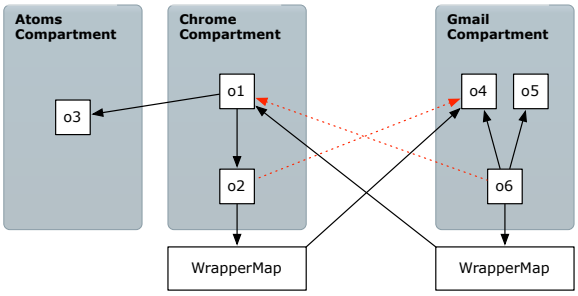

Figure 6. An overview of possible references between compartments.The red arrows represent the old way of communicating between two objects. In the new approach, we add a wrapper objects between 2 objects that reside in different compartments.

tabs) pass objects to each other. Cross-compartment Wrappers are much more than just a remembered set which is common in generational GC environments. Each wrapper is a real Proxy-Object with access-methods that are needed for security restrictions. No direct-communication between compartments is allowed. All communication between compartments must go through these wrapper objects.

When a window or iframe attempts to reference an object that belongs to a different window, we hand 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. For example, if one Google Mail window tries 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 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 much 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 continually performed security checks. This mediation 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 located in different compartments, 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 the 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. For example, if evil.com obtains a reference to a google.com object, we create a wrapper referencing that object in the evil.com compartment. When the 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 handle windows 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. Of course, this action now creates a problem: The outer window might not point directly at the new window, because it is in a different compartment.

We solve this problem using 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 large number 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, since objects from different windows (or tabs) are intermixed on the heap, the browser must walk the entire heap. If a number of idle windows are open, this can be quite wasteful, since those windows have not 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.

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 ignore 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 be collected with only per-compartment GCs since the wrapperMaps would keep them alive.

4.1 Marking

In order to find all reachable objects for a global GC we traverse the object graphs beginning with the 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 hash table followed by marking all global objects.

Alias	URL
280s	280slides.com
AMAZ	amazon.com
BING	bing.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. All sites were visited on January 30th 2011. Some sites required an account in order to perform basic tasks.

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
FLKR	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 when creating new compartments per origin or per iframe.

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 compartment currently performing the GC. Obtaining the compartment identity is done using simple pointer arithmetic and is very cheap as described in Figure 4.

4.2 Sweeping

JavaScript does not support a finalize() method as Java does. However the internal VM design calls a finalize function on every unreachable object during a GC where dynamically allocated memory for an object gets freed. The VM-API also allows overwriting this finalizer function. This is done by the browser and many embedders that use a standalone version of the JavaScript VM.

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 added to the free list 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 in 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” [20].

A web application such as 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. The `src` attribute provides the location of the frame content which is typically an HTML document. 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 listed 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 be beneficial 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.

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 here is that processes offer a stronger separation enforced by the processor hardware, while compartments offer a pure software guarantee. However, compartments benefit by allowing 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 policies), while with processes cross-process JavaScript object access is either impossible or extremely expensive. In the future, browsers 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 all execute in one process.

7. Evaluation

To evaluate our compartmental memory management approach, we implemented it in the open source JavaScript VM SpiderMonkey [18], 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 easy to rerun the benchmarks by setting the `javascript.options.mem.gc_per_compartment` option in the `about:config` page of Firefox. This section uses baseline implementation, called *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 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 must always allocate a new arena if all arenas are full for a certain compartment. This results in higher fragmentation because 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 likely to become garbage. Once there are no reachable objects within the arenas, we can return them to the OS. In the old approach, objects from different domains might have kept arenas alive.

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 with a forced GC in between. 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 a single 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 when they are intermixed 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

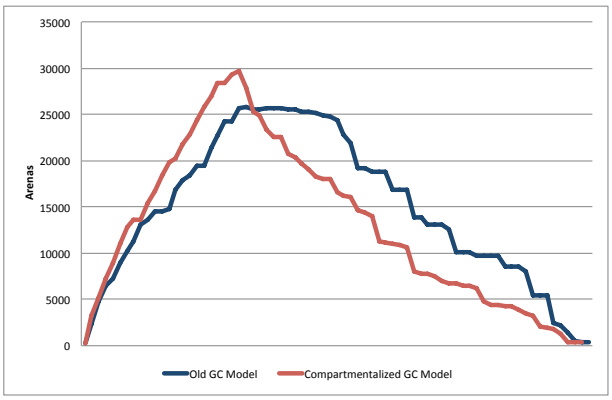

Figure 7. Opening 50 tabs and closing them again with the baseline and per-compartment approach. We can see a higher memory consumption peak for the opening process with the new approach but, once we close tabs, we also deallocate arenas faster.

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. Figure 8 and Figure 9 show 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. *50 Tabs Reachable* represents all objects that are reachable in the JavaScript VM excluding the compartment where the V8 benchmark runs. *V8 Reachable* represents all objects that are reachable within the compartment that triggered the per-compartment GC (V8 compartment). This number also represents the marking workload for the partial GC.

Finalized represents all objects that are finalized during the GC event. *Missed* represents the number of unreachable objects that are not reclaimed because of the per-compartment GC.

Relative values are calculated as follows:

$$Reachable\,Rel. = \frac{V8\,Reachable}{50\,Tabs + V8\,Reachable} * 100\%$$

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

$$Rel.\,to\,Total = \frac{Missed}{Missed + Finalized + 50\,Tabs + V8\,Reachable} * 100\%$$

The first three GCs are global GCs and happen during loading of the 50 tabs. Once we start the V8 benchmark suite we see only compartment GCs until we shut down the browser. The shutdown process performs the last three global GCs.

Reachable Rel. is 0 for the first global GCs. After we start the V8 benchmark suite it is between 0.5% and 1%. For the Earley-Boyer benchmark we see a triangle allocation scheme and Reachable Rel. alternates between 1.5% and 7%. Only during the Splay benchmark, where a huge splay tree is created and modified, does the actual reachable objects within the V8 compartment 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% (Missed Rel.) garbage in other compartments that is not reclaimed. At GC event 37, we see a finalization spike during the Splay benchmark. This indicates that we perform a GC that does not free any memory and we have to increase the heap. The following GC events finalize around 4.5 million objects, but this is not shown due to readability of the graph.

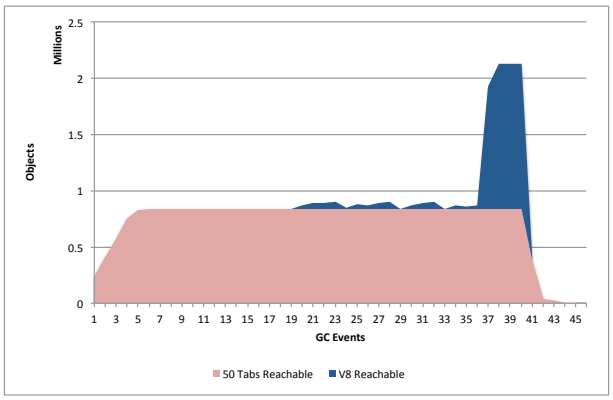

Figure 8. Reachable objects when opening 50 tabs and running the V8 benchmarks. *50 Tabs Reachable* includes all compartments except the compartment where the V8 benchmark runs. *Comp Reachable* means all reachable objects within the V8 compartment.

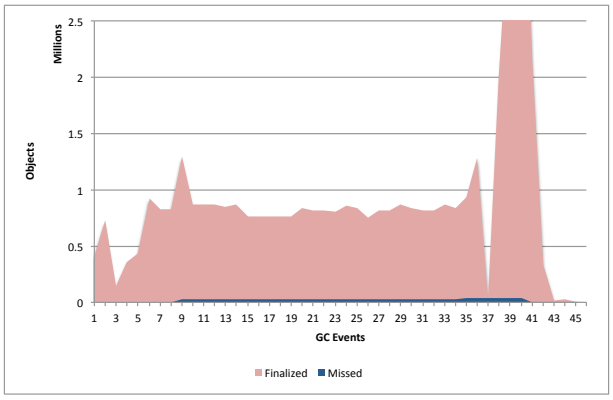

Figure 9. Finalized objects when opening 50 tabs and running the V8 Benchmarks. *Missed* represents the number of unreachable objects that fail to reclaim in other compartments because we only perform per-compartment GC.

Rel. to Total measures the ratio between total heap space and Missed to Finalized. We can see that we only miss to reclaim about 2% of the heap space because of the per-compartment GCs. Note that the number of GC events differs from the results in Table 3 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` [9] in order to measure the duration of each GC event.

Table 4 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 spent in marking increases because we perform more GCs but the average time spent in marking

	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 3. 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.

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

reduces around 12%. The increase in finalization time results from the fact that more objects must be finalized. As explained in Section 4, we also check the mark bit of every single object during sweeping. Since we encounter fewer marked objects and more unreachable objects, the time spent in finalization increases. Marking fewer objects and finalizing more objects indicates a good separation technique.

Figure 10 through Figure 13 show the mark-to-sweep ratio for each GC event for the V8 benchmarks. Figure 10 shows the marking and sweeping ratio for starting the browser, running the V8 benchmarks and closing the browser again with our baseline approach. Figure 11 shows the marking and sweeping ratio with our new per-compartment GC model. We can see that even for a single 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 3 shows that the benchmark score increases from 4505 to 4692 for a single open tab. The big spike almost near the end is caused by the allocation intensive Splay benchmark. The finalization spike at the end is caused by the shutdown of the browser.

The real strength of the new approach shines with many open tabs. Figure 12 and Figure 13 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 12. If we compare this time to Figure 13 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. We can see that the time is identical to the baseline approach for this single spike. Table 3 shows that the benchmark score increases from 3017 to 4511.

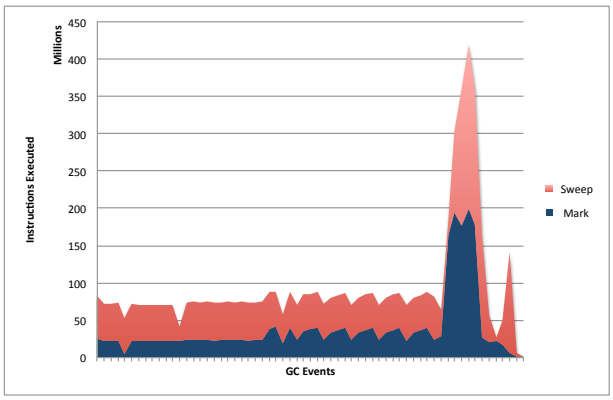

Figure 10. Running the V8 Benchmark Suite with a single tab using baseline approach. The y-axis shows a stacked representation of cycles measured with `rtdsc`.

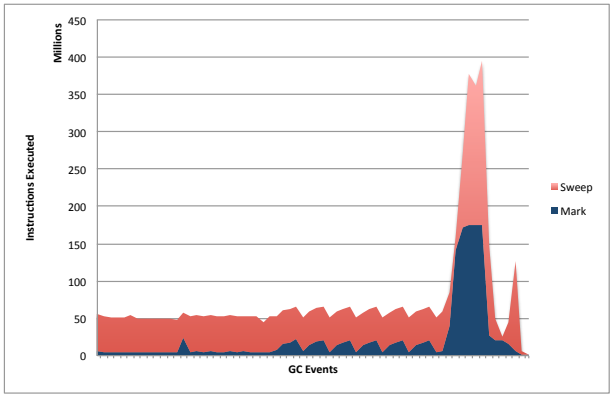

Figure 11. Running the V8 Benchmark Suite with a single tab using per-compartment GC.

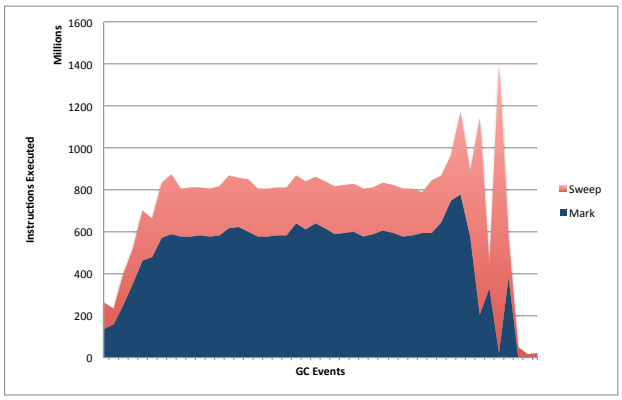

Figure 12. Opening 50 tabs with popular web pages and running the V8 Benchmark Suite using baseline approach. The y-axis shows a stacked representation of cycles measured with `rtdsc`.

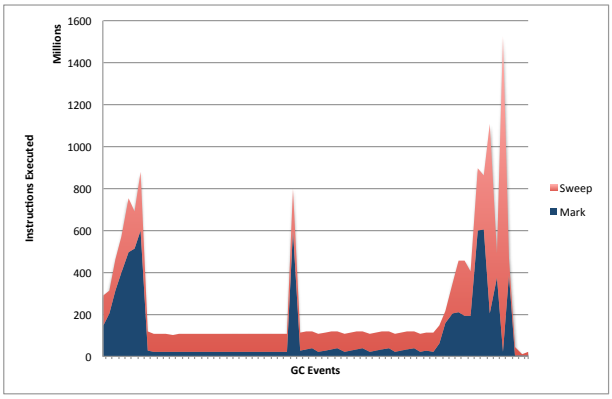

Figure 13. Opening 50 tabs with popular web pages and running the V8 Benchmark Suite using the new per-compartment GC.

Running the benchmark with an additional 50 open tabs now reaches 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 885E6 to 294E6. 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 83%, from 998E6 to 170E6.

7.3 Kraken Benchmarks

Table 5 shows the Kraken benchmark [17] results. The benchmark was executed 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 *Base* column represents the baseline and the *Comp* column represents the per-compartment GC approach. The new approach also introduces more stability for the individual benchmarks. As can be seen in Table 5, the random noise for the individual benchmarks is reduced.

7.4 SunSpider Benchmarks

We claim to 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 near 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 6 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 also has other advantages over 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 [10] 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 [5]. Mark and sweep GC implementations have a long history [13] and we do not claim to reinvent any of the basic ideas. We show how a new layer of abstraction can reduce the workload for such systems and make a real difference for every Firefox user.

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 5. Kraken benchmarks

Our approach can also be described as a simplified version of distributed GC. The Emerald system [11, 12] supports moving objects between physically different nodes. Our solution differs in two ways: All the objects stay in the same process even if they are moved from one compartment to another, and a global GC still performs a GC on the whole heap rather than performing mark and sweep on each individual compartment.

Optimizing allocation patterns to improve the locality of reference in the virtual memory [19] and cache [14] has been studied over many years. Basic implementation like the "first-fit" approach [13] or improvements like the "better-fit" approach [25] still show bad reference locality characteristics. We use object separation based on their origin to obtain better reference locality. For example, internal objects created from chrome code no longer share pages with objects allocated from web sites.

Reis et al. [20] 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 [15] 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. [26] 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. [7] 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 [6] 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 called "partition".

The Beltway [1] system also separates objects in "belts" with the main focus on comparing generational GC aspects.

Benchmark	Base [ms]	Comp [ms]	Speedup [%]
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 6. SunSpider benchmarks.

Seidl et al. [23] 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. [2] 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.

Grier et al. [4] 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. [8] 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. [21] 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.

Our implementation is the default configuration for the current release version 4 of Firefox. It greatly improves the internet experience of several hundred million people every day.

Acknowledgments

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

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0905684. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 153–164. ACM Press, 2002. doi: 10.1145/512529.512548.

[2] 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.

[3] 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.

[4] 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.

[5] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software - Practice and Experience*, 20(1):5–12, 1990. doi: 10.1002/spe.4380200104.

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

[7] 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.

[8] 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.

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

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

[11] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6 (1):109–133, 1988. doi: 10.1145/35037.42182.

[12] N. C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In *Proceedings of the International Workshop on Memory Management*, pages 103–115. LNCS Volume 637, Springer-Verlag, 1992. doi: 10.1007/BFb0017185.

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

[14] 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.

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

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

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

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

[19] J. Peachey, R. Bunt, and C. Colbourn. Some empirical observations on program behavior with applications to program restructuring. *IEEE Transactions on Software Engineering*, 11(2):188–193, 1985. doi: 10.1109/TSE.1985.232193.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.