Thinking Inside the Box: Compartmentalized Garbage Collection

GREGOR WAGNER, Mozilla Corporation
PER LARSEN, STEFAN BRUNTHALER, and MICHAEL FRANZ,
University of California, Irvine

The web browser is the "new desktop." Not only do many users spend most of their time using the browser, the browser has also become host to rich and dynamic applications that were previously tailored to each individual operating system. The lingua franca of web scripting, JavaScript, was pivotal in this development.

Imagine that all desktop applications allocated memory from a single heap managed by the operating system. To reclaim memory upon application shutdown, all processes would then be garbage collected—not just the one being quit. While operating systems improved upon this approach long ago, this was how browsers managed memory until recently.

This article explores *compartmentalized* memory management, an approach tailored specifically to web browsers. The idea is to partition the JavaScript heap into compartments and allocate objects to compartments based on their origin. All objects in the same compartment reference each other direct, whereas cross-origin references go through wrapper objects.

We carefully evaluate our techniques using Mozilla's Firefox browser—which now ships with our enhancements—and demonstrate the benefits of collecting each compartment independently. This simultaneously improves runtime performance (up to 36%) and reduces garbage collection pause times (up to 75%) as well as the memory footprint of the browser. In addition, enforcing the same-origin security policy becomes simple and efficient with compartments.

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

Additional Key Words and Phrases: Garbage collection, isolation, memory management, web-browser architecture

ACM Reference Format:
Gregor Wagner, Per Larsen, Stefan Brunthaler, and Michael Franz. 2016. Thinking inside the box: Compartmentalized garbage collection. ACM Trans. Program. Lang. Syst. 38, 3, Article 9 (April 2016), 37 pages.
DOI: http://dx.doi.org/10.1145/2866576

This material is based upon work supported in part by the National Science Foundation under Grant Nos. CNS-1513837, CNS-0905684, and CCF-1117162 and by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014. 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 Defense Advanced Research Projects Agency (DARPA), its contracting agents, the National Science Foundation, or any other agency of the U.S. Government.
Authors' addresses: G. Wagner, Mozilla Berlin, Haus 10, Treppe 6, Voltastr. 5, 13355 Berlin, Germany; email: gwagner@mozilla.com; P. Larsen, S. Brunthaler, and M. Franz, Department of Computer Science, Donald Bren School of Information & Computer Sciences, University of California, Irvine, Irvine, CA 92697-3435; emails: perl@uci.edu, sbrunthaler@sba-research.org, franz@uci.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0164-0925/2016/04-ART9 $15.00
DOI: http://dx.doi.org/10.1145/2866576

1. MOTIVATION

Since its inception, the Internet has seen rapid growth in terms of users and the number of things these users do on the web. Two decades ago, the web consisted primarily of pages serving static content. Thanks to the ubiquity of JavaScript, webpages now contain highly dynamic content. Consequently, a good user experience not only depends on fast rendering and low network latencies, JavaScript performance matters as well.

Browsers have evolved in response to the changes in usage patterns. For instance, all modern browsers have a tabbed user interface so that users can surf many webpages at once. Also, JavaScript performance has increased substantially thanks to sophisticated just-in-time compilers and optimizations for dynamic languages [Kotzmann et al. 2008; Gal et al. 2009; Hackett and Guo 2012; Hölzle et al. 1991].

Some changes, such as "tabbed browsing," were previously not factored into key browser subsystems, such as the memory manager inside the JavaScript VM. Users with many open tabs or windows inadvertently increase the workload on the garbage collector, which, in turn, leads to noticeable pause times and high memory consumption. Reports collected from the users of the Firefox browser show that some have as many as 200 tabs open; a smaller-scale study found that Firefox power users use up to 25 tabs [Dubroy and Balakrishnan 2010].

Consequently, we motivate our work by observing how scores of the V8 JavaScript benchmark drop when going from 1 to 50 open tabs in Firefox 3.6[1]. Figure 1(a) shows the pause times when running the V8 benchmark suite without having other tabs open. Figure 1(b) shows another run of the V8 suite after opening 50 tabs and loading them with popular websites[2]. Figure 1 shows the garbage collection (GC) pause times when opening 50 tabs and then running the V8 benchmark suite. The x-axis marks GC events; the y-axis shows how the individual mark and sweep phases contribute to the overall pause time. We observe that the score of the V8 benchmark degrades to two-thirds of its single-tab score (from 4511 to 3017 points) as garbage collection times rise. Figure 1(b) also shows that pause times grow linearly with the number of open tabs (events 1–16) and remain high in the benchmarking phase (events 17–66).

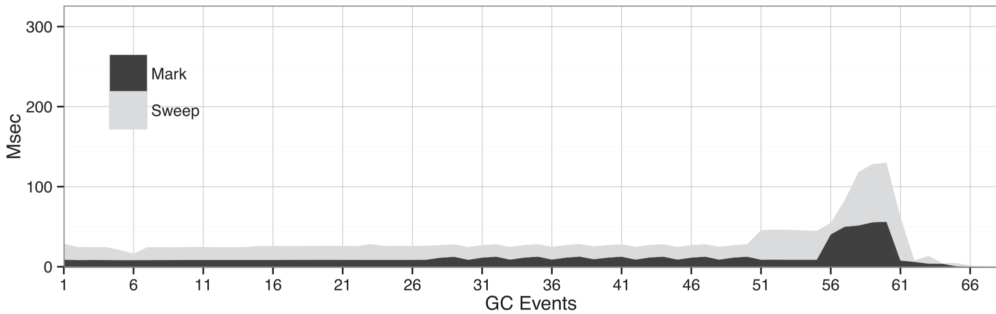
To address this inefficiency, we use domain knowledge to tailor the memory management system to the usage patterns of browsers. Not only does this reduce GC pause times when many tabs are open, it also decreases the overall memory footprint of the browser.

Summing up, we make the following contributions:

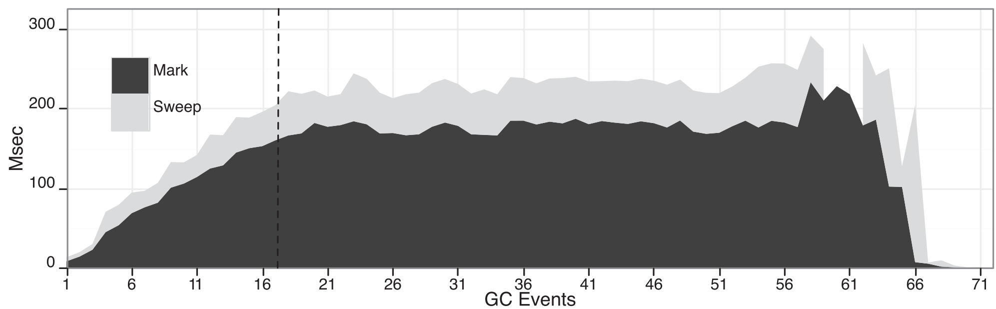
- We introduce the concept of *compartments* into the memory management subsystem (Section 3.1). This design specializes the garbage collector to the usage patterns of browsers. To the best of our knowledge, we are first to present a comprehensive study.
- We identify a good granularity of compartmentalization by comparing alternatives (Section 3.2).
- We optimize our basic approach by adding background finalization and parallel marking (Section 4).
- We perform a careful and detailed evaluation of our compartmentalized memory management approach (Section 5). In particular, our experiments show that:
  - Performance improves by 6% with one open tab and by 36% with 50 open tabs.
  - GC pause times reduce by 69% for a single tab and 75% with 50 tabs.

[1]Our research was implemented in the Firefox browser developed by Mozilla Corporation. The set of enhancements that we developed were first included in releases 4.0, 6.0, and 7.0 (from March to September 2011).

[2]Online Appendix A contains the list of websites used throughout the article.



(a) GC pause times when running the V8 benchmark suite without any other open tabs.



(b) GC pause times when running the V8 benchmark suite after opening 50 tabs with popular web pages. Benchmark events are to the right of the dotted line.

Fig. 1. Opening 50 tabs increases GC pause times.

2. BACKGROUND

An increasing number of activities are taking place on the web, including the exchange of information, goods, and services. The resulting "Internet economy" depends critically on dynamic content and, in turn, on efficient and secure web browsers being developed. In their absence, progress in the area of web application development is artificially stymied.

The addition of JavaScript to the Netscape Navigator 2.0 browser made it easy for developers to change images in response to input events and to validate HTML forms programmatically on the client side. Today, JavaScript is supported by all major browsers and standardized as ECMAScript to allow interoperability. More recently, the emergence of asynchronous JavaScript and XML enables interactive and complex sites that resemble interactive applications rather than static documents. Finally, the move to implement functionality handled by browser plug-ins directly in HTML5 and JavaScript increases the need for efficient JavaScript processing even further.

Historical trends tell a similar story. In the two-year period from November 2010 to November 2012, for example, the average amount of JavaScript code per website grew from 113KB to 215KB [Souders 2013]. This change is likely to continue; Google, Mozilla, and others are developing browser-centric operating systems that run web applications exclusively [Google 2009; Mozilla 2012; Linux Foundation 2012].

The JavaScript language assumes a managed runtime that automatically allocates and frees memory on behalf of the developer. The throughput and pause times of

the garbage collector are therefore major influences on script execution times—and ultimately the user's choice of web browser.

In the mid-1990s, when the use of JavaScript was in its infancy, browsers could deliver acceptable performance using a simple mark-and-sweep collector to manage the JavaScript heap. While such garbage collectors are easy to implement, they may result in subpar performance. If the heap is not partitioned, the collector must scan the entire heap during each collection. This results in comparatively low throughput[3] and high GC pause times. This is particularly detrimental to latency-sensitive applications.

As a result, major browser vendors now use garbage collectors that partition the JavaScript heap to increase performance; for instance, collectors using generational scavenging [Ungar 1984] are common.

Generational collectors partition the heap based on a general observation known as the *weak generational hypothesis*: most objects die young [Hanson 1977; Ungar 1984]. The observation has been confirmed for many languages [Jones et al. 2011]. As objects are allocated, they go into the *nursery* area. Allocation is typically fast, as it can be done by a pointer increment. When the collector runs, reachable objects are copied to an older generation, leaving only garbage, which will be zeroed before it can be reused by the application. By collecting the nursery more frequently than other generations, the average pause times are reduced. Further, since young objects have higher mutation rates than older ones [Huang et al. 2004], co-locating these improves the mutator's locality of reference.

However, since generational collectors copy objects around, they must track the locations of all object references precisely to update the ones pointing to moved objects. Further, they must track intergenerational pointers, since these pointers function as root objects when collecting a single generation. Write barriers are often used to track references into the nursery generation.

A conservative GC, such as the one in Firefox, must be retrofitted with write barriers and switched from conservative stack scanning to a precise GC in preparation for generational collection; this is sometimes infeasible and even unnecessary. We will show that our conservative collector outperforms a state-of-the-art generational GC on some workloads, such as the memory-intensive Splay benchmark in the V8 benchmark suite.

Besides delivering high performance, browsers must safeguard confidential and sensitive information. To that end, browsers employ several types of isolation: JavaScript code executes in a sandbox-mode to prevent interference with applications outside the browser. Additionally, scripts are restricted in terms of what they may access—mainly via the same-origin policy [Rudermann 2001]. In the domain of web content, the origin refers to the (*protocol, hostname, port*)-triplet of any URL. For instance, the origin of

—`http://www.mozilla.com/a.html` equals the origin of
—`http://www.mozilla.com/b.html`

but not to the origins of

—`http://mozillalabs.com/a.html`,
—`http://www.mozilla.com:81/a.html`, or
—**`https://www.mozilla.com/a.html`**

Some web browsers, such as Google Chrome or Microsoft Internet Explorer 8, isolate web content by creating a new process for new tabs or origins. 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

[3]Here, throughput refers to the ratio between the workloads of the collector and the mutator.

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 [Reis and Gribble 2009].

There are two problems with this approach: Certain web features, such as `iframe` navigation, require pages to maintain references to objects belonging to other pages. 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 leads to the allocation of additional memory for thread stacks, relocation tables, and other metadata; the exact overhead varies with the operating system. The use of extra memory per origin is undesirable for environments with limited resources, such as mobile devices.

Since our compartmentalized approach separates the JavaScript heap based on origin, providing isolation in accordance with the same-origin policy becomes simple and efficient (see Section 3.1). Not relying on processes for isolation means that our solution is equally suited for high- and low-end hardware, including phones.

On the other hand, process separation for web applications that avoid cross-origin communication is beneficial in high-end computing environments, as it can prevent programming errors from destabilizing the entire browser. Therefore, we do not view compartmental memory management and multiprocess solutions as mutually exclusive.

3. DESIGN

To set the context for the introduction on compartments, we start by outlining the basic organization of the JavaScript heap prior to our work.

The scripting engine in Firefox, SpiderMonkey[4], is used to congregate all JavaScript objects in a single heap. The heap layout follows a scheme introduced by Hanson [1990]. We allocate memory in 1MB *chunks* from the underlying operating system (Figure 2). A chunk is aligned on 1MB boundaries such that its address can be computed efficiently from the addresses of any object it contains. Chunks are further subdivided into a fixed number of 4 KB *arenas* aligned on 4 KB boundaries. We use the term arena to avoid confusion between virtual memory pages and webpages. In addition to arenas, chunks also contain metadata, including a bitmap used to mark reachable objects during GC.

Each arena consists of a header, padding bytes, and number of slots of a particular size. The header identifies the slot size of the arena, a free list that tracks unused slots and a pointer to the next arena having the same slot size (or the next free arena when the arena is unused). The free-list approach allows efficient allocation; a pointer comparison and increment is all that is required.

Arenas are divided into equal-sized slots such that each arena contains objects of one particular size. Being a dynamic language, the size of a JavaScript object is not fixed at allocation time. Therefore, objects are allocated with an initial size chosen empirically. The memory initially allocated consists of a header and a number of slots. If the number of properties exceeds the number of available slots, a new set of slots is dynamically allocated using `malloc`, and all properties are copied to this memory outside any arena.

Typically, users have several tabs or windows[5] open; eventually one will have allocated enough objects to trigger garbage collection. Since memory from different tabs is interspersed, the marking phase must traverse the entire heap. This is wasteful since

[4]SpiderMonkey is the name of the original interpretive JavaScript engine developed by Mozilla. To increase the execution speed, several JIT compilers targeting frequently executed code have been added. These include TraceMonkey [Gal et al. 2009], JägerMonkey, and IonMonkey [Hackett and Guo 2012]. Since the choice of a JIT compilation strategy is independent of our enhancements to the garbage collection scheme, we equate SpiderMonkey with the entire VM.

[5]We use tabs and windows interchangeably since it makes no difference with regard to memory management.

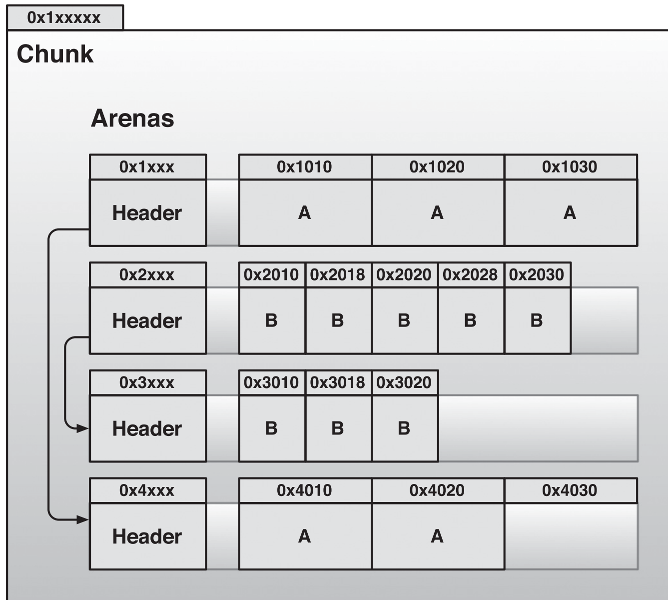


Fig. 2. The JavaScript heap is partitioned into 1MB chunks that are further divided into 4KB arenas. Every arena has a header that stores basic information about the arena. Simple bit arithmetic suffices to compute the chunk or arena header address.

many tabs may be idle and are therefore not producing any garbage to be collected. The fact that the Firefox user interface, or chrome code (not to be confused with Google Chrome), is implemented in JavaScript only exacerbates the problem. Even though objects created by the chrome code typically outlive those allocated by scripts on a page, each collection computes the reachability of these objects.

3.1. Compartments

The key change to the memory-management scheme in the browser is in how JavaScript objects are grouped. As we have already mentioned, generational scavenging groups objects based on age, that is, the number of GCs that they survive. The novelty of our approach is that we partition the heap using a *domain-specific* property—the origin of an object—rather than a generic property that every object has. (Section 3.2 describes why we choose the level of granularity.)

The objects created by `http://gmail.com` and `http://www.bank.com` are placed in two separate compartments, for example. We choose to separate chrome objects (e.g., `chrome://content/browser.xul`) from objects created by webpages. Since chrome objects tend to outlive objects created by webpages, this separation avoids tracing of chrome objects each time a webpage triggers GC. This heuristic is specific to browsers and other applications whose UIs are scriptable (e.g., using JavaScript).

Compartments are stable in the sense that objects are not allowed to change compartments; nor can compartments be split or joined.

We require that cross-origin references between objects go through *wrapper objects* while allowing objects inside the same compartment to reference each other directly (Figure 3). The wrapper objects are stored in a special per-compartment structure called the "WrapperMap." Wrappers predate compartmentalization and are created whenever a script would reference content in another tab or `iframe`; they enforce access control on the underlying object according to the same-origin policy.

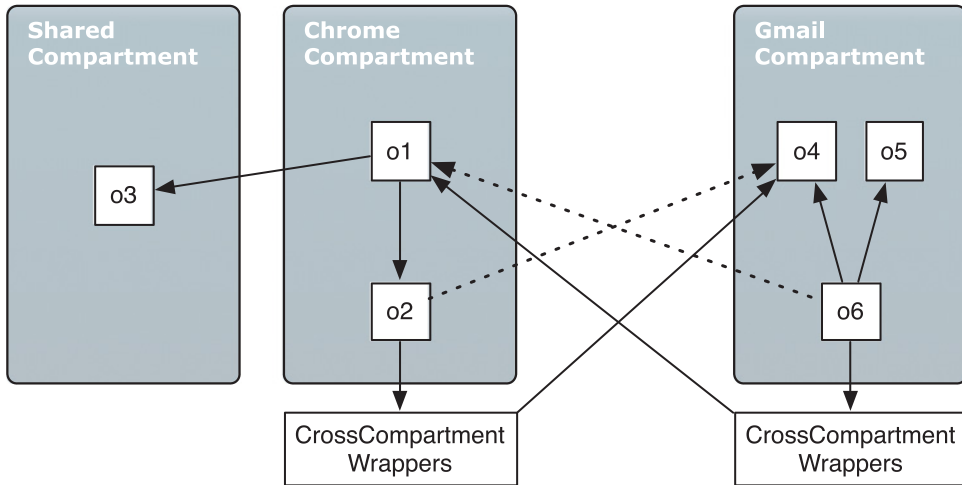


Fig. 3. An overview of possible references between compartments. The dotted arrows represent the old way of communicating between two objects. In the new approach, we add a wrapper object between two objects that reside in different compartments.

The SpiderMonkey VM optimizes the handling of certain[6] strings via *interning*: if two or more scripts contain the same string, each script will reference a single immutable copy of that string. The main benefit is that each interned string is unique; thus, equality testing can be done in constant time by comparing pointers. To facilitate sharing, interned strings (called "atoms" internally) go into a special "atoms compartment." Since referencing an interned string is always permitted, objects in all compartments reference these directly instead of going through wrappers. Because interned strings do not reference other objects, the atoms compartment has no outgoing references. This means that only global GC cycles can reclaim interned strings. Many strings are not interned, however.

Compartmentalization has the following benefits:

(1) We can now perform incremental GC one compartment at a time. Since all outside references to objects inside a compartment go through wrappers, these track the set of objects that are externally reachable and eliminate the need for write barriers (at the expense of a level of indirection for cross-container pointers). When performing a per-compartment GC, we simply walk the object graph inside the compartment while assuming that all externally referenced objects are live; heap areas unrelated to the tab that triggered garbage collection are no longer walked. All objects not marked during the walk are safe to reclaim.
(2) Wrappers are now used in a more principled and efficient way. The wrapper approach of FireFox 3.6 (which mirrors the use of wrappers in other browsers) requires wrappers to be injected manually and at the right places in the C++ code to be secure. With our compartmentalized approach, the JavaScript engine now enforces this invariant uniformly at a low level—leaving less room for error. Wrappers are now private to each compartment, whereas they could previously be accessed by functions from different origins. This means that the origin of the wrapped object as well as the accessing function (which is also an object) in known when it is

[6]The optimization is done at the VM level and remains invisible to users. String literals and identifiers in JavaScript code are typically interned by the parser.

instantiated. Access checks are therefore performed as wrappers are created rather than at access time. The "wrapper factory" simply creates a wrapper that always allows or denies access to the object it contains.

(3) Allocation of new objects does not need synchronization. In the previous design, threads allocated arenas from an arena list and kept them in thread local storage. To avoid data races among multiple threads, the allocation path for arenas was locked. After a GC cycle, arenas would be returned to the shared arena list to be allocated again with locking. Using compartments, we can dispense with almost all locking since arenas stay within a compartment until they are empty and released; the allocation path simply traverses arenas within a compartment without the need for synchronization. Allocation of interned strings still needs fine-grain synchronization, however, since multiple threads may trigger it.

(4) Last, compartmentalization improves locality of reference. To bridge the gap between processor and memory speeds, the CPU loads an entire cache line whenever reads are serviced by the memory subsystem. This is likely to bring more than a single object into the CPU cache. In the old design, objects were adjacent to other objects of the same size regardless of origin. In the new design, objects are only co-located with other objects from the same domain. Since cross-origin objects are used together infrequently, the new design results in higher data-cache utilization and thus higher execution speed.

Per-compartment garbage collection is triggered whenever the allocation level of a compartment reaches a threshold. If the overall allocation exceeds 150%[7] of the triggering compartment's allocation, a global garbage collection is performed instead. (Additional triggers of global GC exist as well, but fall outside the scope of this article.) Even if we wanted to, global GC cycles cannot be entirely eliminated. The reason is that cycles can exist in the global object graph that span multiple compartments. No objects in such a cycle can be reclaimed from collecting any single compartment because the wrapper map would keep them alive; hence, a global walk of the heap is required.

The relationship between compartments, chunks, and arenas is shown in Figure 4. Compartments contain a list of arenas for each size class and the header of each arena was extended with a reference to the compartment to which the arena belongs. Instead of pointing to the next arena with the same size class, each arena header now points to the next arena that holds similarly sized objects *and* that belongs to the same compartment.

Since arenas are no longer shared across origins, we end up consuming more memory. The allocation of a new object can require allocation of a new arena in more situations than in the past. The increased fragmentation is essentially due to the requirement that new memory cannot come from any compartment; the origin of the compartment must match that of the allocating script. In the worst case, the user opens m different pages from different origins and each allocates n objects of different sizes. The objects belonging to each page will partially fill n different 4KB arenas, thus making the worst-case internal fragmentation $O(n \times m)$. On the other hand, the new design often uses direct references for which wrappers would previously have been injected. This partially offsets the increased memory consumption from having private arenas.

Brain Transplants. We finish our overview of our basic approach by discussing an interesting corner case in the design of compartments. Containers of webpages (i.e.,

[7] This heuristic was put in place to avoid performance regressions for mobile devices. If too many compartments allocate memory and we trigger only a single-compartment GC, we would miss freeing all the memory for the other compartments. Low-end devices showed performance regressions when we missed too much memory during a GC.

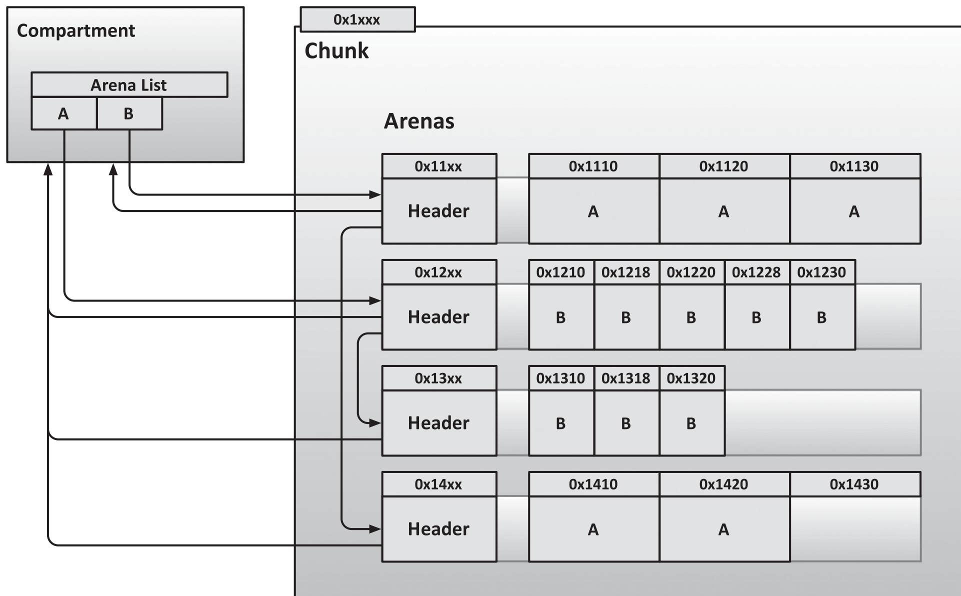


Fig. 4. Compartments keep an arena list which points to the first arena of each size class. Every arena has a header that stores basic information about the arena. It includes a reference to the corresponding compartment.

windows, tabs, and `iframes`) are represented in a somewhat peculiar way in the JavaScript Document Object Model (DOM)[8]. Two objects exist for each DOM window: the inner window and the outer window. The outer window represents the UI element, which acts as a container for the web content that the user sees. The inner window refers to the displayed content itself. This design was implemented as a secure way to handle windows navigating to a new URL.

When the user clicks a hyperlink, the inner window is replaced by a new object, whereas the outer window stays unchanged. Whenever the destination URL has a new origin, we allocate a new inner window in a new compartment as needed. Due to our rules on cross-origin references, the outer window cannot directly reference inner windows with a new origin.

Our solution to this problem—which we call "brain transplants"—copies the outer window into the compartment of the inner window whenever it changes. We keep the object in the old compartment around by transforming it into a wrapper object pointing into the newly created compartment.

3.2. Granularity

It is possible to compartmentalize web content using several granularities; the choice of per-origin granularity is not obvious. One end of the spectrum is represented by the previous design: no compartmentalization at all. The other end of the spectrum is more difficult to define because the boundaries between web programs are unclear. When designing the security architecture of the Google Chrome browser, it was found that "web programs are easy to understand but difficult to define precisely" [Reis and Gribble 2009]. The reason is that any individual web application may contain content,

[8]The DOM is a set of API bindings that allows the structure and contents of HTML documents to be read and manipulated by the JavaScript engine.

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

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

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

scripts and substructures originating from any number of unrelated sources. Further, the user may run several instances of the same web application in different tabs.

If we create too many compartments, we will create more compartment metadata and cross-origin wrappers, which will create a space and performance overhead from increased indirection. If we create too few compartments, we will again end up walking more of the heap for each per-compartment GC cycle; this has an adverse effect on pause times and throughput.

To better understand the trade-off, we carried out measurements with compartmentalization at two different granularities: object separation by origin and separation by the `iframe` in which they are contained. HTML `<iframe>` is an element that can contain another web document, an advertisement, for instance. There is no general way to measure how many `iframes` a webpage has; thus, we choose to compare the two design choices on a list of popular webpages.

We see from Table I that some sites, such as ebay, benefit from the increased granularity offered by separation at the level of `iframes`. However, the number of compartments and wrappers increases significantly for other pages. TechCrunch, for example, leads to the creation of 154 compartments instead of just 11. Similarly, Google Mail, which uses `iframes` extensively, sees wrapper usage increase by a factor of 30 (183 to 5654 wrappers). We conclude that the compartment-per-origin granularity is most suitable and note that it is simultaneously the most natural choice with regard to enforcing the same-origin policy.

4. ENHANCEMENTS

As we will see in the evaluation section, the basic compartmentalized scheme reduces pause times. However, we also recognize several opportunities to optimize the design. Our optimizations offer further improvements to latency and throughput. These are: background finalization, parallel marking, and reduced fragmentation. Wagner describes each of these optimizations in detail [Wagner 2011].

4.1. Background Finalization

The Java language specification exposes a finalization method at the language level and explicitly allows parallel finalization. In contrast, JavaScript does not expose a finalization method at the language level. However, the internal design of the Spider-Monkey VM calls for internal finalization prior to deallocating the memory for each object. In particular, we expose certain object properties to the DOM implementation and let the DOM handle their destruction. Background finalization is therefore specialized to the Firefox rendering engine (Gecko). However, programs that embed the JavaScript VM can override this finalization function.

Hence, the SpiderMonkey GC cycle consists of four steps happening in sequence:

(1) mark reachable objects in each arena,
(2) identify unreachable objects,
(3) finalize and deallocate unreachable objects, and
(4) insert each deallocated object into the free list of its arena.

Among the applications that we analyzed, some spent upwards of 95% of the total GC pause time in the finalization step. This inhibits or degrades certain uses of JavaScript, including games and other interactive applications, which can have high allocation rates of short-lived objects.

We observe that many JavaScript objects can be finalized on a separate thread in the background rather than on the critical path. Objects with a finalizer defined outside the JavaScript VM are still finalized on the critical path, and so are external strings. These objects are in the minority, however, accounting for less than 5% of the finalized objects in our measurements.

When background finalization is ongoing, all arenas in use are locked by the finalization thread. New arenas must be allocated to create new objects concurrent with background finalization. Upon completion, all deallocated arenas are added to the per-compartment list of free objects and signals that allocation of new objects can once again use already allocated arenas.

When two GC cycles happen back to back, the second cannot start before the background finalization associated with the first cycle is complete. Furthermore, in out-of-memory situations, execution will have to wait until free arenas become available before continuing.

4.2. Parallel Marking

Sweeping a single compartment is much faster than sweeping the entire heap. Greatly reducing the sweeping time shifts the bottleneck to the marking step.

The marking phase consists of the following steps:

—Build root set. This is done by conservatively scanning the machine stack, adding global objects and additional roots, including wrapper objects. Root objects are pushed onto the marking stack.
—Once all roots have been identified (or the marking stack is full), mark the set of reachable objects. For each nonleaf object, we push the children objects onto the marking stack.

While both steps allow for parallelization, we focus on the second step of the marking phase; this is particularly profitable for workloads with a big working set. Modifications to the marking stack need to be synchronized to avoid data races. However, data races when marking reachable objects are actually benign: since each thread treats objects identically (setting a mark bit if it is reached) it is safe to mark objects without synchronization.

Our design uses two threads, each having their own marking stack. The main thread collects the root set in its own marking stack. The secondary thread is used only to mark reachable objects and balances the workload with the main thread using a work-stealing approach [Blumofe and Leiserson 1999]. The design can easily scale to an arbitrary number of threads. In our tests, however, using more threads leads to increased overheads. The performance of our benchmarks did not increase past two threads, with the exception of a purpose-built micro-benchmark.

Marking of Rope Strings. Since string objects are among the most frequently used, SpiderMonkey employs several optimizations. A rope string is a string representation developed by Boehm et al. [1995] supporting fast, nondestructive concatenation and practically unlimited string lengths. Ropes represent strings as binary trees in which internal nodes are string headers with no content and whose leaf nodes are either linear strings (character buffers) or rope strings themselves. Delaying copying until a linear string representation is required makes concatenation fast.

The marking of rope strings previously used the Deutch-Schorr-Waite (DSW) algorithm [Schorr and Waite 1967]. The key feature of this approach is its space efficiency. All reachable rope nodes can be marked without using a separate marking stack because storage in the nodes is used to hold the nodes that would be on the marking stack.

Since the graph is mutated during garbage collection, our synchronization-free marking approach is not compatible with this approach. Rather than using locking for marking of rope strings, we switched these to use a regular marking stack approach as well. We think that this is most appropriate for two reasons. First, this allows parallel marking for string-intensive codes as well. Second, the DSW algorithm was developed at a time when memory and processors were almost on parity in terms of speed and memory was a scarce resource. Today, the situation is entirely different. Modern computers and mobile devices have large, but slow, memories and comparatively fast processors. This means that it is usually beneficial to use a little extra memory to avoid the need to write into every node during marking since superfluous writes cause expensive TLB and data-cache misses. In other words, an algorithm that performs well on older, sequential architectures is not necessarily the best for modern, parallel architectures.

4.3. Reducing Fragmentation

An important measure of the quality of a memory management system is how well it uses the memory that it requests from the underlying OS. By reducing fragmentation, we can allocate less memory from the OS.

There are three kinds of fragmentation:

(1) Internal fragmentation represents allocated memory with the main purpose of padding or place holders. This space is wasted and there is no intention to use it. The padding bytes in arenas cause internal fragmentation, for example.
(2) External fragmentation is caused by many small, reachable objects that are scattered over time. A high external fragmentation rate is particularly bad for our memory management system, because it prevents chunks from being returned to the OS.
(3) Data fragmentation occurs when a logical entity is broken into separate pieces that are not stored adjacently. This kind of fragmentation is most relevant to file systems and is not of concern here.

The enhancement that we discuss here is concerned with external fragmentation. A typical usage scenario is that a user continually opens and closes new tabs such that the number of open tabs varies over time [Dubroy and Balakrishnan 2010]. External fragmentation can now cause a situation in which the memory consumption does not

decrease from its high watermark even if the user closes all tabs (see Figure 8(a) in our evaluation).

We identified the sharing of chunks across compartments as the root cause: chrome objects as well as objects belonging to a webpage routinely get allocated to the same chunk. Chrome objects are typically longer lived than web content; thus, even when closing all tabs, each chunk containing just a single chrome object stays allocated. Similarly, interned strings that are shared among all JavaScript code are typically longer lived than objects private to a single origin. Some interned strings are truly immortal and are released from memory only when the browser shuts down. Consequently, chunks containing immortal strings are immortal themselves.

This may be a minor annoyance for users of powerful desktop systems in which unused pages in the virtual memory subsystem are written to secondary storage until needed. However, the situation is entirely different for mobile devices that must perform as little work as possible in the interest of energy efficiency.

Our solution follows our overarching theme of being domain specific. Given that our memory management is specialized to browsers, we know the expected lifetimes of objects: chrome objects and interned strings are longer lived than web content. Consequently, we allocate these objects in separate chunks from those used for web sites. This makes it more likely that objects with similar lifetimes are co-located and thereby increases chances that memory chunks can be released to the OS as tabs are closed.

Algorithm 1 shows a high-level overview of our combined GC enhancements.

5. EVALUATION

We evaluate the improvements described in the earlier two sections—compartments, background finalization, and parallel marking—in terms of space, scalability, and time. We mainly use the synthetic V8 JavaScript benchmarks from Google to demonstrate our approach. We focus on V8 because the SunSpider benchmarks do not allocate enough objects to trigger a GC cycle, and Mozilla's Kraken benchmark was new and unstable at the time that we performed our experiments. V8 consists of seven benchmarks ported from other languages. It contains several applications that create many short-lived objects, thus stresses the memory management subsystem.

All benchmarks were run at least five times; we manually verified that the variance was negligible between runs. The numbers are stable between each run; thus, we report observations from a single run. Additional results from two allocation-intensive applications are given in the appendix.

Since the implementation and evaluation has been ongoing for an extended period of time, our evaluation contains results obtained from two different machines.

Space and Scalability. The evaluation of space and scalability is performed on a Mac Pro workstation having two 2.66GHz Intel Xeon processors with two CPU cores each. The machine also has 4GB RAM running Mac OS X 10.6 and Firefox 4.0 beta 10 with our compartments enabled by default. Our fundamental changes to the memory-management system are too involved to turn off using a compile or runtime flag. We therefore measure the effects before and after applying the patch introducing compartments.

Performance. All performance measurements were carried out on a MacBook Pro having a 2.66GHz Intel Core i7 processor, 8GB RAM running Mac OS X 10.7 and a nightly build of Firefox 7. Due to the significant changes in the SpiderMonkey VM code base done by others, it is not possible to isolate the changes in GC performance between Firefox versions 4 and 7.

ALGORITHM 1: High-Level Overview of GC Enhancements

```
begin Allocate:
    if compartment allocation > watermark then
        if global allocation > 1.5 × watermark then
            Deallocate(global);
        else
            Deallocate(partial);
        end
    end
    if allocating chrome object or interned string then
        allocate in system compartment;
    else
        allocate in compartment for script origin;
    end
end

begin Deallocate:
    Input: scope
    begin Mark:
        ; /* sequentially:                                                  */
        build root set by scanning native and JavaScript stacks;
        add explicit roots and all global objects to root set;
        if scope = partial then
            add cross compartment wrappers to root set;
        end
        ; /* in parallel (using work stealing to balance load):             */
        recursively mark objects;
    end
    begin Sweep:
        ; /* sequentially:                                                  */
        foreach unreachable object with external finalizer do
            run finalizer ;
            deallocate object and insert into arena free list;
        end
        ; /* in parallel with mutator:                                      */
        foreach unreachable object with internal finalizer do
            run finalizer ;
            deallocate object and insert into arena free list;
        end
    end
end
```

Anyone wishing to repeat our experiments can do so by obtaining the proper versions and setting the `javascript.options.mem.gc_per_compartment` option in the `about:config` page in Firefox.

5.1. Space Overhead

We start by examining how compartments affect memory consumption. In our scenario, 50 tabs are opened in sequence and then closed one by one with a forced GC in between (Figure 5). We chose 50 tabs because our test machine became unusable after opening that many pages. The tabs load the websites listed in Online Appendix A. We force a GC after closing each page to determine the memory footprint, as there is no automatic GC after each close.

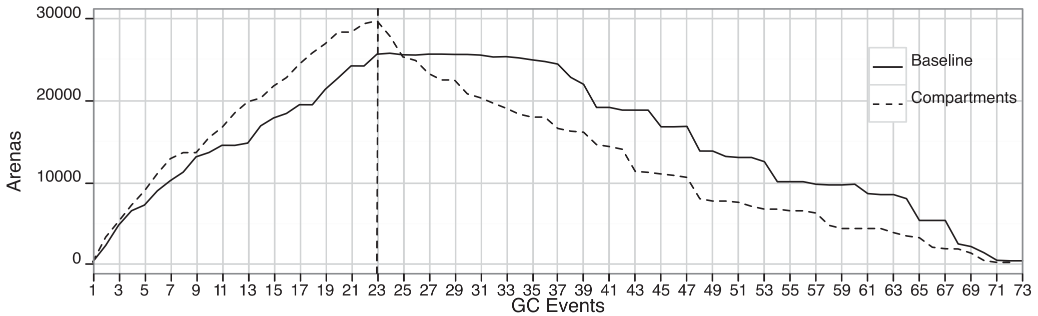


Fig. 5. Opening 50 tabs and closing them again with the baseline and compartment approach. We can see a higher memory consumption peak for the opening process with the new approach, but once we close tabs (to the right of the dotted line), we also deallocate arenas faster.

We see that peak memory consumption is 13% (15MB) higher with compartments. This is expected given that objects from different origins no longer share arenas; this increases fragmentation since we are forced to allocate more arenas (see Section 3.1). We also notice an advantage of the compartmentalized model: memory is released earlier when closing tabs. Because a closed tab releases objects from a given origin, the corresponding arenas become empty.

5.1.1. Effectiveness. Since we are changing the memory-management system design from fully collecting the heap to a partial, per-compartment cycle, another space-related question is how much garbage we miss in each partial collection cycle. We cannot know how much garbage we miss whenever we do a partial sweep. To perform this experiment, it was necessary to change the way our per-compartment sweeping strategy works to record unreclaimed objects across the entire heap. Our solution is the following: whenever a compartment triggers a partial GC, we perform a full GC in a special mode that does not reclaim objects outside the triggering compartment. To emulate a scenario in which many tabs are open and all but a single one remain dormant, we open 50 tabs. Once they are all fully loaded, we start an instance of the V8 benchmark suite. Note that the number of GC events differs from those that we observe when measuring JavaScript performance (e.g., Table IV) since our instrumentation increases pause times, hence affects the benchmark scores.

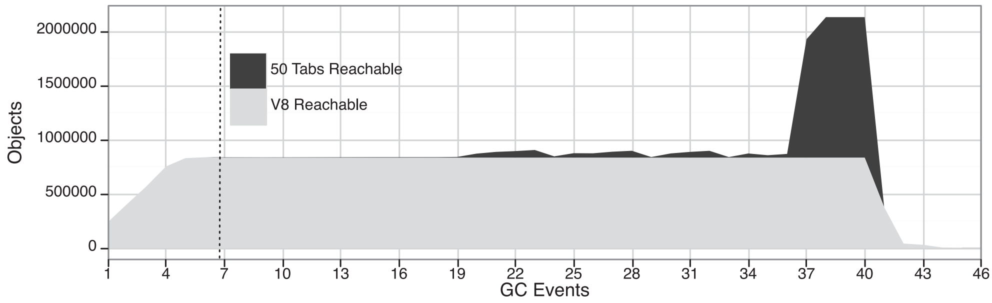
Figure 6(a) shows the results of our experiment. The label *50 Tabs Reachable* represents all reachable JavaScript objects *excluding* those created by the V8 benchmark. The remaining objects are labeled *V8 Reachable* and constitute the marking workload for the per-compartment GC. Figure 6(b) shows the number of unreachable objects that are deallocated as well as the ones that are missed by a partial GC. In both figures, the first three GC cycles that happen as tabs are opened are global. Once the V8 benchmark suite is running, we see only per-compartment GGs until three global GCs are triggered as the browser terminates.

To quantify the storage reclamation efficiency of compartmentalized GC relative to full GC, we define the following metrics:

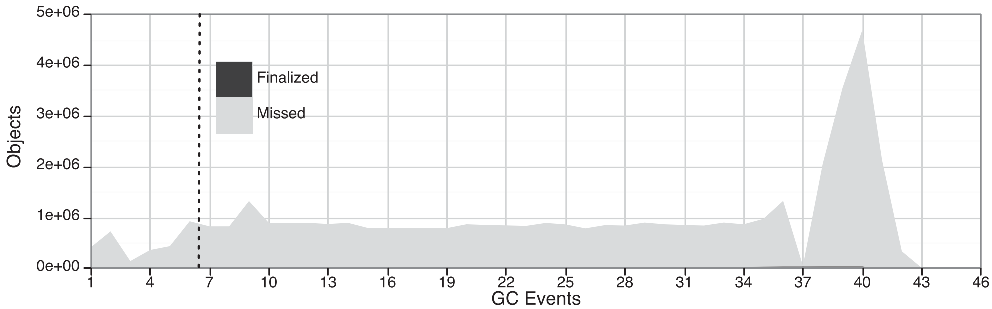
$$\text{Reachable fraction} = \frac{\text{V8 Reachable}}{\text{50 Reachable + V8 Reachable}}$$

$$\text{Missed fraction} = \frac{\text{Missed}}{\text{Missed + Finalized}}$$

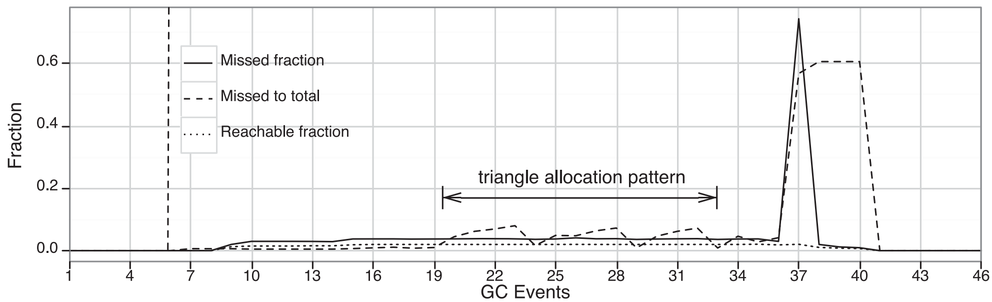
$$\text{Missed to total} = \frac{\text{Missed}}{\text{Missed + Finalized + 50 Tabs Reachable + V8 Reachable}}$$



(a) 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. *V8 Reachable* means all reachable objects within the V8 compartment.



(b) Finalized objects when opening 50 tabs and running the V8 benchmarks. *Missed* represents the number of unreachable objects that fail to be reclaimed in other compartments because we only perform per-compartment GC.



(c) Reachable fraction, missed fraction and missed to total fraction when opening 50 tabs and running the V8 benchmarks.

Fig. 6. Reachable and finalized objects when running the V8 benchmarks. The dotted lines separate the opening of tabs from the benchmarking phase.

After we start the V8 benchmarks, less than 1% reachable objects belong to the V8 compartment (Reachable fraction in Figure 6(c)). For the `EarleyBoyer` benchmark, we see a triangle allocation scheme and a jump to 8% reachable objects in the compartment. Only during the `Splay` benchmark, which creates and modifies a huge splay tree, does Reachable fraction represent 60% of the entire browser heap.

The number of objects that are missed due to partial rather than full GC (Missed fraction) create at most 4% garbage in other compartments that remain unreclaimed during benchmarking. At GC event 37, we see a sharp drop in the number of objects deallocated during the `Splay` benchmark. This indicates that the GC did not free any memory; thus, the heap must increase in size. The following events deallocate 4.5 million objects, as shown in Figure 6(b).

Finally, we measured the number of missed objects relative to the total number of objects that populate the browser heap (Missed to total in Figure 6(c)). The fraction of objects that we do not reclaim due to compartmentalized GC never exceeds 2% of the total object count. In summary, we find that the effectiveness of the garbage collector is largely unaffected by compartmentalization.

5.1.2. Fragmentation. Recall that memory is allocated from the operating system in 1MB chunks. Chunks are subdivided into 4KB arenas and are not released until all arenas are released. We know from experience that chrome objects tend to live on longer than objects created by web content. To reduce fragmentation, we avoid mixing long-lived objects with short-lived ones within a single arena.

To measure the impact of fragmentation, we load 50 tabs with webpages, then close the tabs one by one. Figure 7 shows how fragmentation is reduced using a heap visualization tool implemented by Nicholas Nethercote from Mozilla. Each row represents a 1MB chunk, and each cell inside is a 4KB arena. Colors are used to distinguish compartments (with reuse due to limited colors). Empty or unused arenas are white.

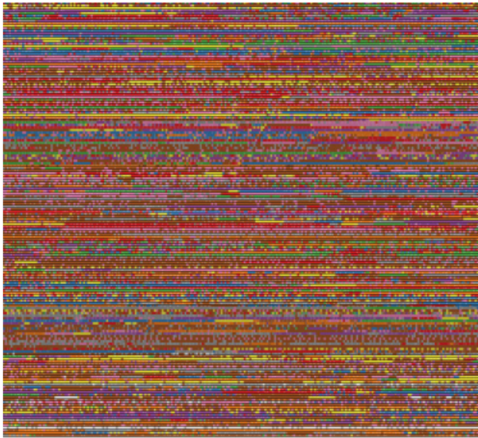
Figure 7(a) shows the heap when all 50 tabs are open. The mixing of colors show that each chunk contains arenas for several compartments. After closing all the tabs (Figure 7(b)), we see that most chunks are nearly empty but must stay allocated. When we avoid mixing long- and short-lived objects (Figure 7(c)), we see that each row contains fewer distinct colors, hence fewer distinct arenas. After closing all the tabs again (Figure 7(d)), most of the chunks have been deallocated and the remaining ones pack arenas more densely.

Figure 8(a) shows the number of chunks, arenas, and compartments over time (per GC event) before optimizing fragmentation. We see that, even though the number of compartments and arenas decrease, the number of chunks (thus, memory consumption at the OS level) stays essentially constant. We see from Figure 8(b) that separating objects based on their expected lifetimes releases memory to the OS as soon as the compartment for each tab is collected.

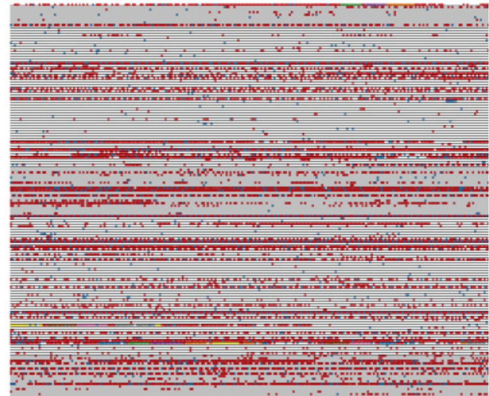
The reduced memory footprint is particularly beneficial to users of resource-constrained systems such as laptops and mobile devices.

5.1.3. Scalability. As explained in Section 3.1, our compartmentalized design affects peak memory consumption; for example, we saw a 15% increase over the baseline design when opening 50 tabs. Browsers that isolate web content using operating-system processes also see an overhead from the resources and metadata allocated for each process. Firefox, which uses a single process model, does not incur this overhead. We therefore compare the space overhead from compartments with the overhead from process isolation, as it is implemented in the Chrome canary version from July 2011 [Reis and Gribble 2009] and Opera 11.50. Note that, since the implementations of the three browsers diverge beyond the design of their garbage collectors, we cannot isolate the impact of the GC design.

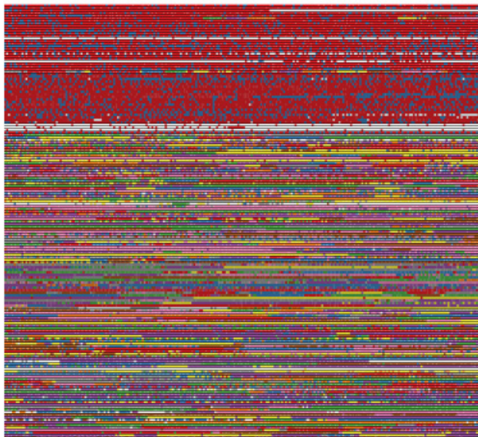
For this experiment, we use JavaScript code that opens 150 websites, one after another. We close all windows except one, and close the browser afterwards. The results are measured with the UNIX `time` command, and can be seen in Table II. `Real` shows the elapsed wall clock time in seconds, `user` shows the total number of CPU seconds



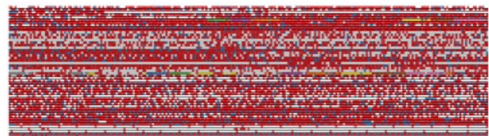
(a) JS heap snapshot for 50 open tabs without compartments.



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



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



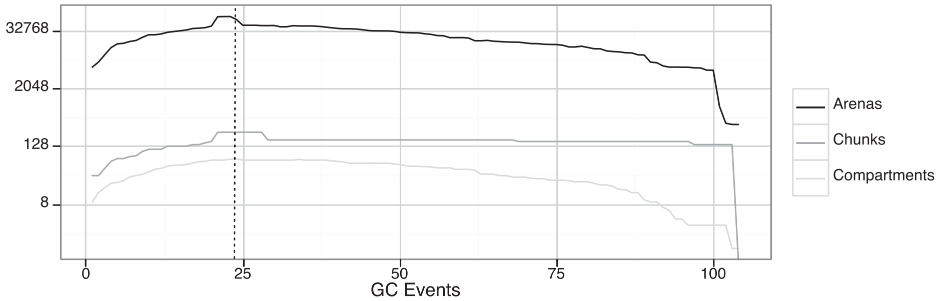
(d) JS heap snapshot for opening and closing 50 pages with the new approach. We see that the system chunks remain but all user chunks are returned to the OS.

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

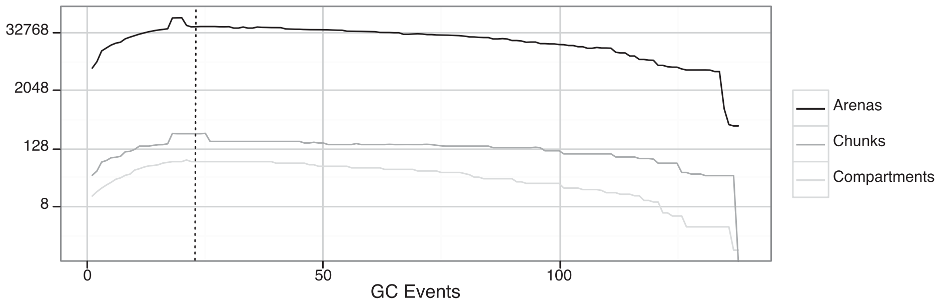
that the process spent in user mode, and sys shows the total number of CPU seconds that the process spent in kernel mode.

We see that Firefox with our modifications and Opera both take little more than 6min to open all websites, whereas Chrome spends almost 30min. The Chrome user experience degrades after about 70 pages and the browser barely responds after opening all the pages. Presumably, the observed slowdown is due to excessive disk paging activity; increasing the amount of physical memory in the system would ameliorate this. Firefox, in contrast, remains fully responsive and scrolling is smooth as well.

We show two columns for Chrome due to the following observation. In our experiments, we ended up with far fewer processes than expected, just three: the renderer



(a) We open 50 websites and close them again. We can see that the number of arenas decreases over time but the number of allocated chunks is constant.



(b) We open 50 websites and close them again with separation of user and system chunks. The number of allocated chunks reduces over time.

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

Table II. Scalability Test

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

process, the main Chrome process, and a helper process. It turns out that programmatically opened pages do not spawn new processes. A documented workaround[9] allowed us to spawn processes for new pages, creating the effect of manually opening the pages. The column labeled *Chrome II* shows the measured times with proper process isolation. We observed 43 Chrome renderer processes in addition to the main and helper processes. Chrome's self-reported memory consumption (`about:memory`) slightly exceeds 5GB, and the browser is unresponsive; in comparison, Firefox consumes 2GB.

We also noticed that Chrome reported an uneven mapping from sites to processes. Some processes apparently host 2 to 3 sites, while others host about half of the sites due to a bug in Chrome's task manager[10].

The main Chrome process contains 368 threads with 150 tabs open and as many as 420 during browser shutdown. Even in the presence of inaccurate reporting of memory consumption, we can directly observe that the system does not scale. Near the end of

[9]http://dev.chromium.org/developers/design-documents/process-models.
[10]http://code.google.com/p/chromium/issues/detail?id=91757.

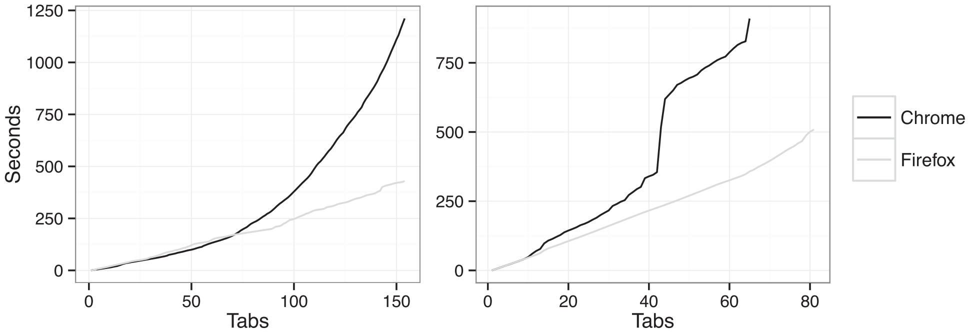


Fig. 9. Wall-clock time required to open 150 tabs in Firefox and Chrome. The graph on the left shows the performance on a high-end laptop with 8 GB RAM whereas the one on the right was obtained on a netbook having only 1 GB of RAM. Note the different scales.

the test, we cannot help but notice that opening a new tab takes far too long. In 2011, a bug report was opened based on our finding[11].

Figure 9 shows the detailed scaling behavior of Firefox and Chrome on two different systems: a laptop in which RAM is plentiful (left) and a memory constrained netbook (right). We see that Firefox shows essentially linear scaling behavior, whereas the curve for Chrome is exponential after about 70 tabs. On a netbook, Firefox scales linearly until 80 sites are open; after that, we see warnings that scripts are no longer responsive. In comparison, Chrome stops rendering pages correctly after 40 tabs and several seconds.

Finally, we compared our approach with the Opera browser. We find that it offers comparable speed while having a higher peak memory consumption (2.5GB vs. 2GB for Firefox).

5.2. Performance

Several well-known JavaScript benchmarks are available, including SunSpider from Apple, Kraken from Mozilla, as well as V8 and Octane from Google. We focus on the results that we obtained using the V8 benchmark in this section; results from SunSpider, Kraken, and two JavaScript-intensive web applications are found in Appendices A and B, respectively.

The V8 benchmark is rate-based, meaning that it measures completed runs of a benchmarking kernel within a certain time frame (1s). Fixing the time during which each benchmark is run means that the amount of memory allocated varies with the execution speed. Speeding up an allocation-intensive benchmark causes it to allocate more objects within the same amount of time.

We performed VM internal measurements to correlate the GC events with the V8 benchmark workloads. We use the `rdtsc` [Intel 1997] instruction to measure the duration of GC events with high accuracy.

From our motivating example, we know that the number of tabs open affects the JavaScript performance markedly before the introduction of compartments. Therefore, we divide our experiments into two sets: a single-tab scenario and a multiple-tab one.

5.2.1. Single-Tab Performance. Table III shows the benchmark scores when no additional tabs are open. With compartments, we still benefit from separation of objects belonging to the user interface, strings, and the contents of the tab. We see that the introduction of compartments increases the benchmark score by 2% on its own (4872 to 4983) due to

[11] https://groups.google.com/a/chromium.org/group/chromium-dev/browse_thread/thread/ba8c629ec7f7096c#.

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

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

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

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

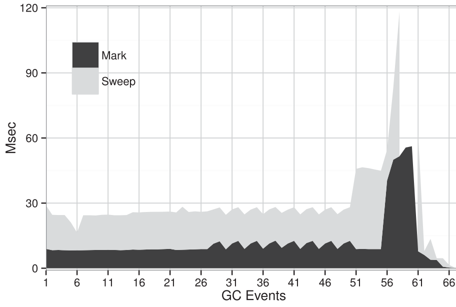
	Base	Compartments	Background finalize	Parallel marking	Percent reduction
Marking [ms]	805	474	522	370	54
Sweeping [ms]	1323	1265	236	241	82
Finalize objects [ms]	1072	1051	221	228	79
Finalize strings [ms]	224	200	0	0	100
Total [ms]	2240	1823	837	690	69

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

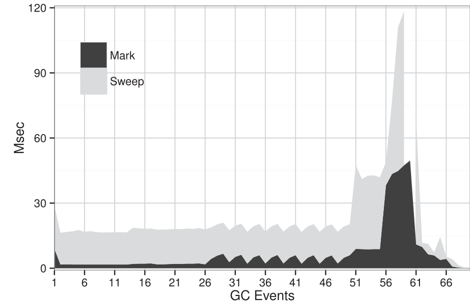
better performance in memory-intensive benchmarks such as `Raytrace`, `EarleyBoyer`, and `Splay`. Background finalization increases the performance by an additional 3% (4983 to 5154). Again, the performance increases are highest for allocation-intensive codes. We see only a slight overall increase in performance from parallel marking. One benchmark, `DeltaBlue`, sees a 2% slowdown. In summary, the aggregate effect of our optimizations increase overall performance by 6% and the performance of allocation-intensive codes by as much as 14%.

Table IV shows the average GC pause times. We observe that the overall number of GC events increased due to compartmentalization (6% in line with the performance gains). The ability to mark a single compartment decreases the marking time by 54% (805ms to 370ms).Many features within Firefox are implemented in JavaScript and allocate objects in the systems compartment. Therefore, compartmentalization is beneficial even when only a single tab is open. The sweeping time dropsfurther due to background finalization: 82%. Objects with external finalizers still contribute 220ms to the sweeping phase. All strings, however, are now finalized in the background. The aggregate reduction of pause times on the V8 benchmark in a single-tabbed scenario is 69%. The basic compartmentalized design is responsible for 19% of the overall improvement. Background finalization and parallel marking account for 54% and 18% of the pause-time reduction, respectively.

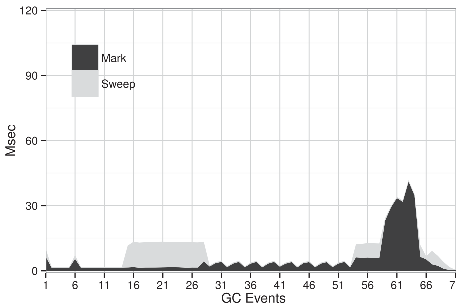
We graph the individual pause times of each GC event when running the V8 benchmark in Figure 10. We see that, without compartments (Figure 10(a)), marking contributes about one-third to the overall pause, whereas sweeping contributes two-thirds. The pause time hovers round 20ms until it peaks at 120ms for the `Splay` benchmark. We see that, with the basic compartment approach (Figure 10(b)), the marking time decreases while the sweeping time remains mostly the same. Adding background finalization dramatically reduces sweeping time; parallel marking reduces the marking time further (Figure 10(c)).



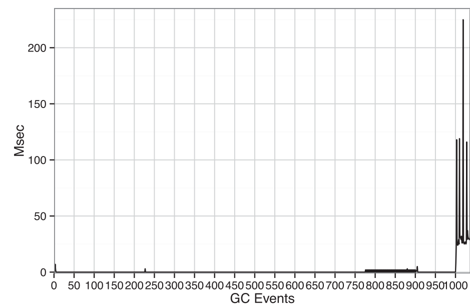
(a) V8 Benchmark with baseline approach in a single tab.



(b) V8 Benchmark with compartment approach in a single tab.



(c) V8 Benchmark with parallel marking approach in a single tab.



(d) GC events for the V8 Benchmark within the Chromium Browser.

Fig. 10. Single-tab GC pause times with four different GC schemes.

When comparing with the Chromium browser (Figure 10(d)), which uses generational scavenging to manage the heap, we observe low pause times during most benchmarks. (Several GC events reportedly take 0ms.) In part, the reason is that Chromium GC cycles are much more frequent. As a result, Chromium spends a factor of 2.5 more time on GC compared to Firefox (1837ms vs. 690ms). Moreover, the `Splay` benchmark shows peak pause times of up to 240ms on Chromium; this is about 6 times longer than the peak GC latency with compartments.

5.2.2. Multiple-Tab Performance. We repeated our experiments in a setting intended to emulate a user who keeps many tabs open in addition to a JavaScript-intensive page (V8). In particular, we started the browser and loaded 50 webpages, starting measurements once all pages were loaded. As the motivating example in Section 1 demonstrated, this is a situation in which the baseline memory-management approach does not perform well.

We start by looking at the individual benchmark improvements listed in Table V. In comparison to Table III, we see that the baseline score drops by more than one-third, from 4872 to 3082 by keeping 50 tabs open. The memory-intensive benchmarks—`Raytrace`, `EarleyBoyer`, and `Splay`—run significantly slower, which identifies memory management as the bottleneck. In particular, `EarleyBoyer` drops to less than half of its single-tab score.

Compartmentalization increases the scores by 33%, on average, and by a factor of 2.3 in the case of `EarleyBoyer`. Background finalization increases performance further by about 5% on allocation-intensive benchmarks—indicating that finalization is not so

Table V. V8 Scores with 50 Open Tabs and One Tab Running the V8 Benchmark Suite

	Base	Comp	Background Finalize	Parallel Marking	Relative
Richards	7216	8137	8127	8035	10%
DeltaBlue	2543	4860	4311	4540	44%
Crypto	7351	8594	8024	8495	13%
RayTrace	1864	3594	3899	3860	52%
EarleyBoyer	1815	4198	4449	4517	60%
RegExp	1377	1526	1569	1613	15%
Splay	4213	5664	6169	6275	33%
Score	3083	4611	4652	4795	36%

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

Table VI. Basic Internal Measurements for the V8 Benchmark

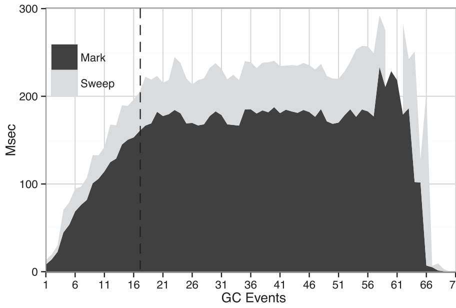
	Base	Comp	Background Finalize	Parallel Marking	Relative
GC Events	49	66	68	69	41%
Marking [ms]	7711	1641	1651	1366	82%
Sweeping [ms]	2821	1923	913	873	69%
Finalize objects [ms]	2219	1610	802	764	66%
Finalize strings [ms]	303	207	6	6	98%
Total [ms]	11998	4342	3448	3024	75%

Note: With the new compartment approach and all optimizations, we reduce the time spent in the GC during the V8 benchmark by 75%. Relative compares the base approach with the parallel marking approach. Smaller scores are better.

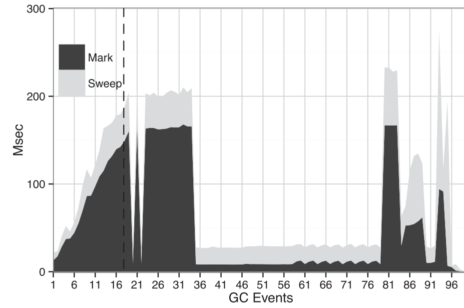
much of a bottleneck with multiple open tabs. Similarly, parallel marking increases performance by an additional 3%. Overall, the aggregate impact of our enhancements on the V8 benchmark score is 36% when 50 additional tabs are open. By comparing with Table III once more, we see that the benchmarking score with 50 tabs open is on par with the single-tab performance without compartments and about 7% slower than single-tab benchmarking with compartments.

Again, we measured how the individual phases in GC cycles contributed to the overall pause time (Table VI). First, we see that the baseline approach spends almost 8s to mark reachable objects versus just 805ms in the single-tab scenario. The basic compartment design reduces marking time back down to about twice the single-tab time (1641ms). The residual increase in marking time is due to the remaining global GC events, as we will see when we graph the GC events. Overall, we see marking times reduced by 89% and sweeping times reduced by 69%, which leads to an an aggregate improvement of 75%. In terms of wall clock time, the cumulative GC pause time goes from 12s to just 3s.

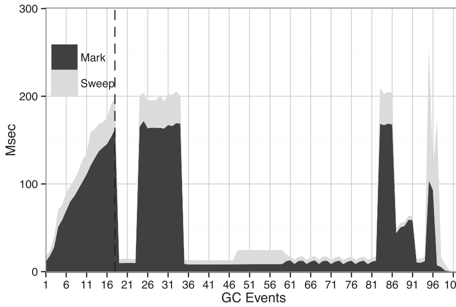
In Figure 11(a), we see that running the V8 benchmark with 50 additional tabs open using the baseline design results in pause times lasting 230ms, on average, and up to 340ms under the `Splay` benchmark. With compartments, GC pause times follow a bimodal distribution (Figure 11(b)). Global GC cycles, which appear during loading of the 50 pages (events 1–16), when running `RegExp` and during shutdown, are not any faster than with the baseline approach. The `RegExp` benchmark allocates large arrays outside the JavaScript heap. Since the heap size is not tracked on a per-compartment level, a runtime-wide watermark is passed. This triggers a global collection. Looking at the per-compartment cycles, we see that these are markedly faster; they take 30ms in general and 130 during the `Splay` benchmark. Adding background finalization (Figure 11(c)) reduces the sweeping times, as expected. Finally, we add parallel



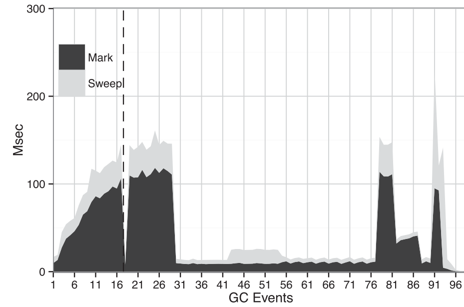
(a) V8 Benchmark with baseline approach and 50 other tabs.



(b) V8 Benchmark with compartment approach and 50 other tabs.



(c) V8 Benchmark with background finalization approach and 50 other tabs.



(d) V8 Benchmark with parallel marking approach and 50 other tabs.

Fig. 11. Multiple-tab GC pause times with four different GC schemes. The dotted lines separate the opening of 50 tabs from the benchmarking phase.

marking (Figure 11(d)). This optimization has the greatest impact on the time to perform global GCs; their durations decrease from about 230ms to 150ms as a result of global marking times dropping from around 170ms to 100ms.

6. RELATED WORK

There is a wealth of research on the design and implementation of automatic-memory management; Jones et al. [2011], Wilson [1992], and Zorn [1989] all offer good overviews.

While this article builds on many previous works within the field, we restrict our discussion to the most important ones. In the interest of clarity, we discuss work related to each individual addition separately: basic compartmentalization, enhancements to finalization, then parallel marking. Finally, we describe how other modern browsers handle memory management and isolate web content.

6.1. Compartments

Currently, the Firefox memory management system uses mark-and-sweep GC. In particular, the way that the heap is divided into regions for fast allocation is due to Hanson [1990], who showed that explicit regions in C could achieve the fastest known heap allocation mechanism based on time performance per allocated byte.

Regions are also known as arenas, zones, generations, partitions, and compartments. The idea of partitioning the heap to increase locality and decrease the workload of the

garbage collector has been studied in great detail. The key challenge is to find a good partitioning heuristic; properties such as age, stack, and object connectivity have been used to map objects to partitions. The most successful scheme, generational scavenging [Ungar 1984], is based on the weak generational hypothesis—the expectation that most objects die young [Hanson 1977].

The basic mark-and-sweep approach is also well known [Knuth 1973]; we do not claim that this work is novel because we extend it. Rather, we describe an alternative partitioning of the heap into regions and a new way to map objects to these—both are distinguished by virtue of being *domain specific* rather than *generic*.

As early as 1967, Ross [1967] created a library that allocates memory to different zones. Each zone has its own allocator and can be deallocated all at once. Vmalloc [Vo 1996] extends this idea by providing more flexible allocation and deallocation interfaces. The allocation policy is per region and includes memory layout, type, and allocation algorithm.

Hayes [1991] observed that some objects act as roots to entire clusters of objects. Dependent objects are likely to be unreachable if their roots are; thus, examining the roots more frequently than other objects can reduce the work of the garbage collector. Similarly, our compartmentalized approach seeks to focus on the region (cluster) of the heap that has seen the most allocation activity since the last collection cycle, and saves work by ignoring all other compartments.

Barrett and Zorn [1993] present a region-based system in which they predict the lifetime of objects during allocation. They use a profile-based predictor that segregates short-lived objects from long-lived ones. Objects that are predicted to be short-lived are allocated in small spaces by incrementing a pointer and deallocated together when they are all unreachable.

Aiken et al. [1995] use static analysis to replace runtime GC with compile-time annotations. All memory allocation and deallocations are placed explicitly in the program.

Grunwald et al. [1993] present a performance evaluation of the reference locality of dynamic storage-allocation algorithms. Similar to our findings, they show that the design of a memory allocator can significantly affect the reference locality, and therefore lead to an increased number of page faults and cache misses.

Tofte and Talpin built a region inference system [Tofte and Talpin 1997; Tofte 1998]. They use a type inference system that specifies where regions can be allocated and deallocated. They further specify to which region each allocation site writes.

Berger et al. [2002] study custom allocators in applications that use explicit memory management. They find that the use of custom allocators is often outperformed by the use of a high-performance, general-purpose memory allocator with the exception of region-based allocators. They propose a new abstraction, *reaps*, that generalizes region and heap abstraction by allowing both single-object and bulk-memory reclamation.

6.2. Heap Partitioning

Hudson and Moss [1992] developed a generational GC toolkit in which the mature object space is divided into multiple fixed-size areas to allow incremental collection. Like our compartments, a remembered set tracks references into each area so that each area can be collected separately. Collecting just one area during each scavenging cycle bounds the time that the collector spends on the mature object space.

Appel [1989] and Barrett and Zorn [1995] also address the problem of reclaiming tenured garbage more effectively. They dynamically adjust the boundary between young and old objects to achieve optimal memory usage. Tenured objects can even move to the untenured space again by a boundary adjustment. The Beltway system

[Blackburn et al. 2002] also separates objects in "belts" with the main focus on comparing generational GC aspects. It, too, features configurable partitions.

The hoard memory allocator [Berger et al. 2000] partitions application memory into a global heap and P thread-local heaps. The heaps are divided into superblocks and blocks similar to our use of arenas and chunks. Each thread can allocate memory only from its corresponding heap. This design reduces false sharing of cache lines and bounds the "blowup," that is, the difference between maximum memory consumption under serial and parallel executions of the same program.

Hirzel et al. [2002] analyze the connectivity of heap objects. First, they show the importance of understanding heap connectivity of objects. Second, they provide hints on improving existing partitioning models. While the research is focused on Java, the focus on connectivity remains relevant for JavaScript as well. Similar to our work, they seek to co-locate objects with similar lifetimes and access patterns inside a partition.

Domani et al. [2002] research thread-local heaps for Java. Each thread is assigned partition of the heap, to which it allocates its objects. Write barriers guard accesses from other threads, and objects known to be shared are allocated in a shared-heap partition. They allow thread local GCs so that the collection of local objects is synchronization-free; as with our approach, global collections remain necessary to reclaim shared objects. Steensgaard [2000] also studies thread-specific heaps. Partitioning, coupled with escape analysis to detect thread-private objects, lowers the workload of thread-specific GC cycles and allows allocation to happen without synchronization—mirroring how we are able to eliminate locking from the allocation path.

Seidl and Zorn [1998] build a profile-driven object lifetime and access frequency predictor. They use it to 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.

Previous works use properties such as age, size, sharing, and connectivity to allocate objects to partitions. These are all generic properties, and can be determined or estimated for every object irrespective of the application domain. In contrast, we use the provenance of objects created by scripts—an attribute that is specific to web content and browser chrome code. By customizing our solution to the application that we target, we can dispense with complex static analysis or dynamic profiling to determine which objects are likely to be used together because it follows from our understanding of the web-browsing domain.

Researchers have proposed static analysis [Guyer and McKinley 2004], dynamic profiling [Chilimbi and Larus 1998], changes in the traversal algorithm [Wilson et al. 1991], online object reordering [Huang et al. 2004], and allocation heuristics [Shuf et al. 2002] to improve cache utilization. Our approach achieves good locality of references naturally as a side effect of separating objects based on their origin because cross-origin references are rare.

Our approach is conceptually similar to a simple distributed garbage collector [Abdullahi and Ringwood 1998]. Besides being confined to a single machine, the key differences are as follows. Even if objects change compartment (on navigation), the containing process never changes. This is in contrast to distributed approaches, such as the Emerald system [Juul and Jul 1992; Jul et al. 1988], which can move objects between two physically distributed nodes. Second, we retain the ability to perform a global GC across the entire heap.

Our basic compartmentalized memory-management system, which shipped in Firefox 4.0, was previously published by Wagner et al. [2011]. It does not describe our subsequent enhancements to compartmentalization, that is, background finalization, parallel marking, and separation of objects based on expected lifetimes.

6.3. Finalization

The idea of finalizing objects on a separate thread is also used in the Microsoft .NET environment [Richter 2000]. Unlike our background finalization solution, which remains fully hidden from JavaScript developers, .NET developers must know when and how to properly use the exposed `Finalize` method. The problem is that Microsoft UI frameworks follow a long-standing convention that only the thread that created a UI object may finalize it. This is inherently at odds with background finalization since objects created by a UI thread are thereby prevented from being finalized on another thread. In particular, the .NET finalization thread must send a message to the UI thread to dispose of an object that it created. The finalizer thread stalls as long as the UI thread is busy; thus, performance quickly deteriorates for applications that allocate many objects requiring finalization on the UI thread.

This problem is related to our background finalization constraint: objects created outside SpiderMonkey are finalized on the main thread. Since we finalize objects of this kind on the main thread, the thread that processes objects with VM-internal finalizers does not need to synchronize its execution with other threads.

In Java, support for background finalization is part of the language itself. The documentation [Oracle 2010] for `java.lang.Object.finalize` states that:

> *The Java programming language does not guarantee which thread will invoke the finalize method for any given object.*

Background finalization can be compared to lazy sweeping [Zorn 1989; Hughes 1982]. The latter performs sweeping on demand as new allocations are requested. The allocator traverses unswept memory objects, sweeps the first suitable object it finds, and returns it to satisfy the memory request. Lazy sweeping thereby distributes the sweeping cost among allocation sites. Reallocating an object shortly after it is swept also increases spatial locality. The Safari browser from Apple uses this strategy.

6.4. Parallel Marking

The prevalence of dual and multicore machines has allowed parallel GC algorithms to enter the mainstream. Parallel marking is now supported by all major JavaScript VMs. Blelloch and Cheng [1999] and Cheng and Blelloch [2001] present a parallel real-time garbage collector. Like our system, they use a conservative stack-scanning approach. They split large objects such as arrays so that they may be marked in parallel.

Endo et al. [1997] and Flood et al. [2001] describe parallel, nonconcurrent collectors that use load balancing. Like our implementation, work stealing [Blumofe and Leiserson 1999] is used to distribute the work evenly among the threads.

Siebert [2008] discusses the limits of parallel-marking GC. Parallel-marking algorithms perform poorly when only some of the available processors can perform scanning work. Processors pick the next element out of the gray set—if the gray set is empty, the processors must stall. An example with poor marking performance is a linked list with only one root. Our parallel-marking approach uses a single-helper thread that starts when the first root objects are discovered. The main thread pushes all roots on the marking stack and the background thread starts marking them.

Barabash and Petrank [2010] also study the scalability of parallel tracing collectors. In particular, they investigate the heap depth (an object at depth d must be dereferenced d times sequentially, thus limiting the available parallelism) of standard Java benchmarks and show that several produce heap shapes with low scalability. They also suggest two measures to extract additional parallelism from such heap shapes: heap shortcuts into long, narrow structures such as linked lists; and speculative tracing on idle processors starting at random heap objects.

6.5. Isolation and Fragmentation

Soman et al. [2006, 2008] discuss performance issues with multitasking managed runtime environments for Java. They find that the memory-management subsystem performs poorly compared to their single-tasking counterparts. Similar to our approach, they introduce heap isolation and use heap space efficiently to lower the multitasking overhead. Their multitasking memory manager performs generational collection such that the young generations of applications are swept individually; intergeneration references are remembered with a card-making scheme. GC cycles consist of marking reachable objects, selecting regions for evacuation, evacuation itself, and sweeping. After marking, regions are selected for evacuation based on the amount of fragmentation caused by sweeping.

Lang and Dupont [1987] partition the heap and support both mark-and-sweep and copying collection. During collection, a partition to be compacted is chosen at random; the remaining ones are swept.

Siebert [2000] studies fragmentation in a noncopying GC and introduces a new object model in the Jamaica JVM. Like chunks and arenas in our design, the heap is divided into fixed block sizes from which objects are allocated; large objects are handled specially using a linked list of noncontiguous blocks. By choosing the proper block size for each benchmark program, overall fragmentation is greatly reduced.

Johnstone [1997] and Johnstone and Wilson [1998] shows that fragmentation costs for most programs are insignificant. His real-time, noncopying generational garbage collector uses a write-barrier optimization to deliver hard real-time behavior.

Reducing fragmentation is also related to reducing the cost of copying in generational GC. Researchers have shown that information available during allocation may be used to reduce the number of objects that are copied during GC [Blackburn et al. 2001].

6.6. Competing Browsers

The designs of modern browsers are not discussed extensively in the academic literature as their development is largely taking place outside universities and research labs. Out of necessity, we resort to information from developer documentation, blog posts, and bug reports maintained by the developers.

Chrome. Reis and Gribble [2009] discuss the boundaries of web applications and how these are reflected in the design of the Chrome browser from Google. 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 Chrome. In contrast to our work, they attempt to create new processes for new domains. Section 5.1.3 shows how the Chrome (Chromium) process model compares to compartment-level isolation in terms of scalability.

Currently, Chrome's V8 JavaScript engine employs a stop-the-world, generational collector to focus the effort on the youngest part of the heap [Google 2012]. As we saw in Section 5.1.3, the pauses created by Chrome's generational collector are mostly short; but, in the worst case, they can be much longer than with compartments (see Figure 10(d)). Since surviving objects are promoted into the old generation, the V8 engine must track the location of all object references precisely such that they may be updated when the referenced objects move.

Internet Explorer. Microsoft [2008] also uses OS processes to isolate content in different tabs 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`.

Internet Explorer 9 shipped with the Chakra JavaScript engine, which makes extensive use of parallelism to improve performance and shorten GC pause times [Miadowicz 2012]. Microsoft describes the memory-management system as a "conservative, quasi-generational, mark-and-sweep" garbage collector that runs mostly concurrent on a dedicated thread. A second point of similarity is related to string handling. As with our atoms compartment, Internet Explorer dedicates a part of the heap to string objects (and numbers) since these cannot point to other objects and need not be marked.

Safari. Apple builds its Safari browser atop the WebKit layout engine, which also happens to be used by Google's Chrome browser. Following the introduction of process-level separation in Chrome, WebKit 2.0 received built-in support for a split-process model in which the sandboxed web content runs in a separate process from the privileged UI code. Like Chrome, Safari uses a generational garbage collector [Apple 2012].

Opera. Although details are scarce and the code proprietary, the Carakan JavaScript engine in the Opera browser also seems to employ a scheme that partitions the heap based on object provenance. Each document loaded in a tab or an `iframe` gets its own partition [Lindström 2009]. Rather than using wrappers, Carakan detects when object references cross partitions and merge two heaps into one when (permissible) accesses would otherwise cross partitions. The Opera developers seek the same benefits as we do: each collection is faster since it traverses only part of the heap and the JavaScript performance does not degrade as more tabs are opened. Our scalability experiments show that Opera and Firefox with compartments perform comparably. This result reinforces our expectation that the techniques that we present benefit not just Firefox but other browser implementations as well.

7. CONCLUSIONS

This article demonstrates that a custom memory-management strategy can outperform state-of-the-art approaches (such as generational scavenging) by exploiting domain knowledge. Our hypothesis is supported by our implementation that is now part of the popular Firefox browser.

The usage model of browsers has changed to the point that a rethinking of the GC strategy was necessary to keep up with the increasingly JavaScript-intensive web content and the popularity of tabbed browsing. Our solution—partitioning the heap into compartments—addresses both of these challenges and facilitates further optimization. In contrast to approaches that isolate web content using many OS processes, our approach easily scales to 150 open tabs without degrading the user experience; hence, our techniques apply equally well to regular PCs and mobile devices.

In the single-tab experiments, our combined techniques increase V8 benchmark scores by 6%, on average, and as much as 14% on allocation-intensive codes. Further, we observe that pause times are reduced by 69%, on average.

In scenarios in which users keep many (50) tabs open, our improvements are even more pronounced. We report a 36% increase in V8 benchmarking scores and a 75% reduction in pause times.

By gradually integrating our techniques into the mainline Firefox code over three major releases, several hundred million users have already benefited from this work in two ways. First, the reduced pause times enable a wider range of latency-sensitive applications to be built with web technologies. Second, the JavaScript performance with many tabs open is now sufficiently close to single-tabbed performance levels; thus, users need not artificially limit the number of open tabs.

Perhaps the strongest testament to the relevance of compartmentalization is that Mozilla is continuing the effort on its own. For example, scripts have access to global

Table VII. Kraken Benchmarks with and without Per-Compartment GC Optimization. For Each Run

Benchmark	Base [ms]	+/− [%]	Comp [ms]	+/− [%]	Speedup [%]
astar	1236.3	5.0	1182.7	5.8	1.05
beat-detection	457.0	12.9	418.5	3.4	1.09
dft	496.8	13.2	473.5	3.6	1.05
fft	343.2	13.5	348.6	3.7	0.96
oscillator	290.8	0.7	290.8	0.7	1
gaussian-blur	492.5	0.2	492.0	0.2	1
darkroom	221.7	0.5	221.0	0.2	1
desaturate	487.6	5.1	477.1	0.2	1.02
parse-financial	131.3	32.7	111.8	1.3	1.17
stringify-tb	96.2	51.2	71.8	2.3	1.34
aes	231.6	23.4	234.8	9.4	0.99
ccm	154.5	2.1	161.3	8.2	0.96
pbkdf2	313.4	23.8	237.6	7.4	1.32
sha256[a]-it	198.2	39.0	95.9	2.7	2.07
Total	5151.1	1.8	4817	1.7	1.07

Note: We report the mean Runtime and the 95% Confidence Interval.

objects describing the current request, session, window, and so on. Establishing a one-to-one mapping between global objects and compartments simultaneously enables the SpiderMonkey JITs to generate better machine code and simplifies the design of the VM itself. Or, as one enthusiastic developer put it[12]: “having one compartment per origin was an amazing milestone, having one compartment per object [...] is a never-ending stream of goodness.”

APPENDICES

A. ADDITIONAL JAVASCRIPT BENCHMARK RESULTS

While the bulk of the performance evaluation uses the Google V8 benchmarks, we also measure the performance impacts on the older SunSpider suite from Apple and Mozilla’s more recent Kraken suite.

A.1. Kraken Benchmarks

Table VII shows the Kraken benchmark [Mozilla 2011] results. The benchmark was executed in a browser while having 50 websites loaded, replicating the multiple tab V8 benchmark run from Section 5.2.2. We observe 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 reduces the variability of the benchmark scores.

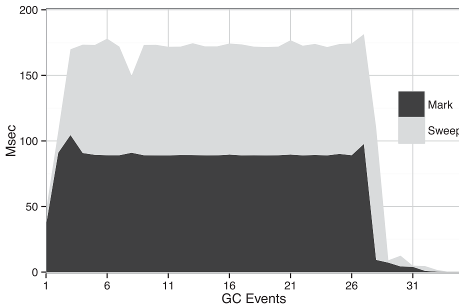
A.2. SunSpider Benchmarks

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 no GC events occur during benchmarking. Also, the reduced synchronization 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 in which actual execution time is measured. Table VIII shows the results of the SunSpider benchmark suite. (We do not

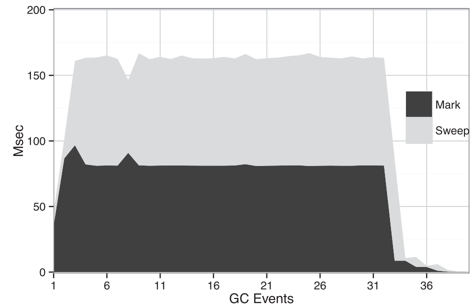
[12]http://bholley.wordpress.com/2012/05/04/at-long-last-compartment-per-global/.

Table VIII. SunSpider Benchmarks with and without Per-Compartment GC Optimization

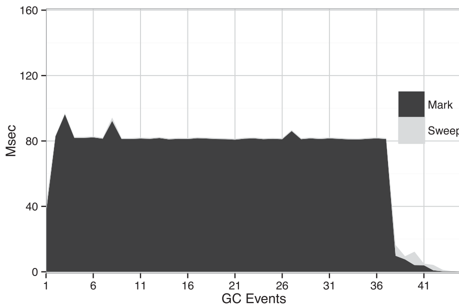
Benchmark	Base [ms]	Comp [ms]	Speedup [%]
cube	16.1	15.7	2.48
morph	16.1	15.8	1.86
raytrace	36.5	36.2	0.82
binary-trees	19.9	19.1	4.02
fannkuch	13	12.9	0.77
bitwise-and	1.3	1.2	7.69
nsieve-bits	4.3	4.2	2.33
recursive	21.3	20.9	1.88
aes	10.5	10.3	1.90
md5	5.3	5.2	1.89
format-tofte	20.1	19.4	3.48
format-xparb	13.4	12.8	4.48
cordic	8.3	4.6	44.58
partial-sums	7.9	7.8	1.27
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



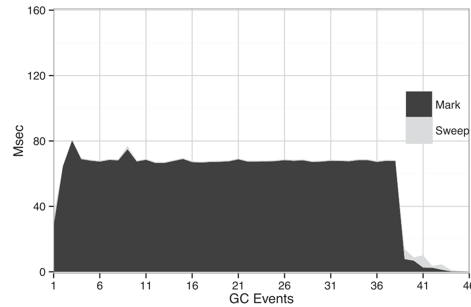
(a) Raytracer animation with baseline approach.



(b) Raytracer animation with compartment approach.

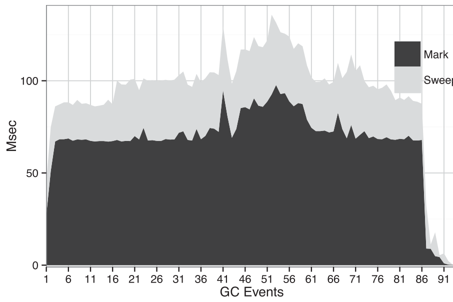


(c) Raytracer animation with background finalization approach.

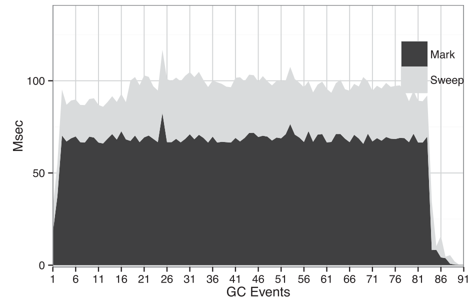


(d) Raytracer animation with parallel marking approach.

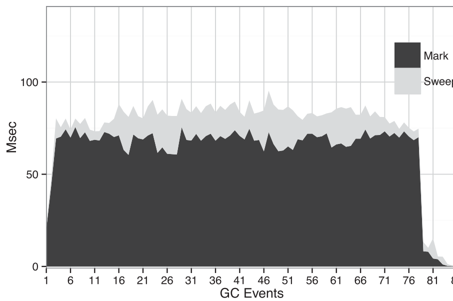
Fig. 12. GC pause times for a real-time ray tracer.



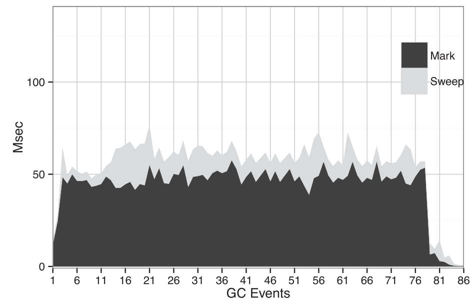
(a) Flight of the Navigator animation with baseline approach.



(b) Flight of the Navigator animation with compartment approach.



(c) Flight of the Navigator animation with background finalization approach.



(d) Flight of the Navigator animation with parallel marking approach.

Fig. 13. GC pause times for a browser-based flight simulator.

show six benchmarks that do not show any change: nbody, nsieve, 3bits-in-byte, bits-in-byte, sha1, and spectral-norm, but include them when calculating averages). We see a 3% overall improvement with our compartmentalized scheme.

B. NONSYNTHETIC WEB WORKLOADS

Reducing the GC pause time not only helps increase synthetic benchmark scores, our new approach also greatly improves the performance of all allocation heavy web applications 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.

B.1. Real-Time Ray Tracer

Figure 12(a) shows the GC events for a real-time ray tracer. We can see that equal amounts of time are spent in marking and sweeping. The average GC pause time is about 175ms, which results in noticeable pauses during the animation. Single-compartment GC reduces the overall GC pause time to about 160ms by not including the objects allocated in the system compartments during the GC, as can be seen in Figure 12(b). A more pronounced improvement comes from background finalization. Figure 12(c) shows that the average GC pause time decreases to about 80ms. Finally, Figure 12(d) shows the result with the parallel-marking approach on top of background finalization. The average GC pause time reduces to about 60ms. There is still a little

hiccup during the animation, but the user experience improves significantly in comparison to the original 175ms pause time.

B.2. Flight of the Navigator

The Flight of the Navigator animation uses the modern WebGL API for hardware-accelerated graphics and stresses the memory-management part of the SpiderMonkey VM. Figure 13(a) shows the baseline situation for the animation. We see that the animation contains peak pause times of up to 140ms, which results in noticeable jumps within the animation. Switching to the compartmentalized approach (Figure 13(b)) reduces the average GC pause time to about 100ms. Most of the GC cost comes from marking the data structures that the animation allocates in the JS heap. Background finalization reduces the average pause time further to about 85ms (Figure 13(c)). The marking cost of the GC becomes the major bottleneck. Adding parallel marking on top of background finalization reduces the pause time further, to about 60ms (Figure 13(d)). These are similar results to the real-time ray tracer (Section B.1). A pause time of 60ms does not eliminate all stutter, but is nevertheless a big improvement over the pause times that we see with the baseline approach.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors thank Jason Orendorff and Blake Kaplan from Mozilla, who worked hard in order to make this research happen. Michael Bebenita and Mason Chang from UC Irvine gave valuable feedback. Other important help came from members of the Mozilla community. They tested our approach, reported bugs, and helped us address them. Finally, we thank the anonymous reviewers and Mark Murphy for their constructive feedback and suggested improvements to draft versions of this article.

REFERENCES

Saleh E. Abdullahi and Graem A. Ringwood. 1998. Garbage collecting the Internet: A survey of distributed garbage collection. *Computing Surveys* 30, 330–373. http://doi.acm.org/10.1145/292469.292471.

Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM Press, New York, NY, 174–185. DOI:http://dx.doi.org/10.1145/207110.207137

A. W. Appel. 1989. Simple generational garbage collection and fast allocation. *Software—Practice and Experience* 19, 2, 171–183. DOI:http://dx.doi.org/10.1002/spe.4380190206

Apple. 2012. The WebKit Open Source Project. (2012). Retrieved February 29, 2016 from http://www.webkit.org/projects/javascript/index.html.

Katherine Barabash and Erez Petrank. 2010. Tracing garbage collection on highly parallel platforms. In *Proceedings of the International Symposium on Memory Management (ISMM'10)*. ACM, New York, NY, 1–10.

David A. Barrett and Benjamin G. Zorn. 1993. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*. ACM Press, New York, NY, 187–196. DOI:http://dx.doi.org/10.1145/155090.155108

David A. Barrett and Benjamin G. Zorn. 1995. Garbage collection using a dynamic threatening boundary. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM Press, New York, NY, 301–314. DOI:http://dx.doi.org/10.1145/207110.207164

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM Press, New York, NY, 117–128.

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering custom memory allocation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*. ACM Press, New York, NY, 1–12.

Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. 2002. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, New York, NY, 153–164. DOI:http://dx.doi.org/10.1145/512529.512548

Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*. ACM Press, New York, NY, 342–352. DOI:http://dx.doi.org/10.1145/504282.504307

Guy E. Blelloch and Perry Cheng. 1999. On bounding time and space for multiprocessor garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. ACM Press, New York, NY, 104–117. DOI:http://dx.doi.org/10.1145/301618.301648

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46, 5, 720–748.

Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An alternative to strings. *Software—Practice and Experience* 25, 12, (December 1995), 1315–1330.

Perry Cheng and Guy E. Blelloch. 2001. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM Press, New York, NY, 125–136. DOI:http://dx.doi.org/10.1145/378795.378823

Trishul M. Chilimbi and James R. Larus. 1998. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management (ISMM'98)*. ACM, New York, NY, 37–48. DOI:http://dx.doi.org/10.1145/286860.286865

Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-local heaps for Java. In *Proceedings of the International Symposium on Memory Management (ISMM'02)*. ACM Press, New York, NY, 76–87. DOI:http://dx.doi.org/10.1145/512429.512439

Patrick Dubroy and Ravin Balakrishnan. 2010. A study of tabbed browsing among Mozilla Firefox users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'10)*. ACM Press, New York, NY, 673–682.

Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. 1997. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *SC*. ACM, New York, NY, 48.

Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. 2001. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium (JVM'01)*. USENIX Association, Berkeley, CA, 21–21. http://portal.acm.org/citation.cfm?id=1267847.1267868.

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM Press, New York, NY, 465–478. DOI:http://dx.doi.org/10.1145/1542476.1542528

Google. 2009. Introducing Google Chrome OS. Retrieved February 29, 2016 from http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html.

Google. 2012. Chrome V8 - Efficient Garbage Collection. (2012). Retrieved February 29, 2016 from https://developers.google.com/v8/design#garb_coll. Accessed 01/31/2013.

Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*. ACM Press, New York, NY, 177–186. DOI:http://dx.doi.org/10.1145/155090.155107

Samuel Z. Guyer and Kathryn S. McKinley. 2004. Finding your cronies: Static analysis for dynamic object colocation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*. ACM, New York, NY, 237–250. http://doi.acm.org/10.1145/1028976.1028996

Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM Press, New York, NY, 239–250.

David R. Hanson. 1977. Storage management for an implementation of SNOBOL4. *Software—Practice and Experience* 7, 2, 179–192. DOI:http://dx.doi.org/10.1002/spe.4380070206

D. R. Hanson. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience* 20, 1, 5–12. DOI:http://dx.doi.org/10.1002/spe.4380200104

Barry Hayes. 1991. Using key object opportunism to collect old objects. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*. ACM Press, New York, NY, 33–46. DOI:http://dx.doi.org/10.1145/117954.117957

Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. 2002. Understanding the connectivity of heap objects. In *Proceedings of the International Symposium on Memory Management (ISMM'02)*. ACM Press, New York, NY, 36–49. DOI:http://dx.doi.org/10.1145/512429.512435

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*. Springer, Berlin.

Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot, B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: Improving program locality. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM Press, New York, NY, 69–80.

Richard L. Hudson and J. Eliot B. Moss. 1992. Incremental collection of mature objects. In *Memory Management*, Yves Bekkers and Jacques Cohen (Eds.). Lecture Notes in Computer Science, Vol. 637. Springer, Berlin, 388–403. DOI:http://dx.doi.org/10.1007/BFb0017203

R. J. M. Hughes. 1982. A semi-incremental garbage collection algorithm. *Software: Practice and Experience* 12, 11, 1081–1082. DOI:http://dx.doi.org/10.1002/spe.4380121108

Intel. 1997. Using the RDTSC Instruction for Performance Monitoring. Retrieved February 29, 2016 from http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

Mark Stuart Johnstone. 1997. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. Ph.D. Dissertation. University of Texas at Austin, Austin, TX. AAI9824978.

Mark S. Johnstone and Paul R. Wilson. 1998. The memory fragmentation problem: Solved? In *Proceedings of the International Symposium on Memory Management (ISMM'98)*. ACM, New York, NY, 26–36. DOI:http://dx.doi.org/10.1145/286860.286864

Richard Jones, Anthony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC & Sons, Inc., London, UK.

Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems* 6, 1, 109–133. DOI:http://dx.doi.org/10.1145/35037.42182

Niels Christian Juul and Eric Jul. 1992. Comprehensive and robust garbage collection in a distributed system. In *Proceedings of the International Symposium on Memory Management*, Lecture Notes in Computer Science, Vol. 637. Springer, Berlin, 103–115. DOI:http://dx.doi.org/10.1007/BFb0017185

Donald E. Knuth. 1973. *The Art of Computer Programming, Fundamental Algorithms* (2nd ed.). Vol. 1. Addison Wesley, Boston, MA.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java hotspot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization* 5, 1, 7:1–7:32.

B. Lang and F. Dupont. 1987. Incremental incrementally compacting garbage collection. In *Papers of the Symposium on Interpreters and Interpretive Techniques (SIGPLAN'87)*. ACM, New York, NY, 253–263. DOI:http://dx.doi.org/10.1145/29650.29677

Jens Lindström. 2009. Carakan Revisited. Retrieved February 29, 2016 from https://dev.opera.com/blog/carakan-revisited/.

Linux Foundation. 2012. Tizen. Retrieved February 29, 2016 from https://www.tizen.org. Accessed 02/01/2013.

Andrew Miadowicz. 2012. Advances in JavaScript Performance in IE10 and Windows 8. Retrieved February 29, 2016 from http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx.

Microsoft. 2008. What's New in Internet Explorer 8. Retrieved February 29, 2016 from http://msdn.microsoft.com/en-us/library/cc288472.aspxhttp://msdn.microsoft.com/en-us/library/cc288472spx.

Mozilla. 2011. Kraken JavaScript Benchmark. Retrieved February 29, 2016 from http://krakenbenchmark.mozilla.org/.

Mozilla. 2012. Experience Firefox OS on your Android device. Retrieved February 29, 2016 from http://www.mozilla.org/en-US/firefoxos/. Accessed 02/01/2013.

Oracle. 2010. Java 2 Platform SE v.1.4.2 API Specification. https://docs.oracle.com/javase/1.4.2/docs/api/. Accessed 01/28/2013.

Charles Reis and Steven D. Gribble. 2009. Isolating web programs in modern browser architectures. In *Proceedings of the European Conference on Computer Systems*. ACM Press, New York, NY, 219–232. DOI:http://dx.doi.org/10.1145/1519065.1519090

Jeffrey Richter. 2000. Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework. http://www.cs.inf.ethz.ch/37-201/files/GC_in_NET.pdf. Accessed 01/28/2013.

Douglas T. Ross. 1967. The AED free storage package. *Communications of the ACM* 10, 481–492. Issue 8. DOI:http://dx.doi.org/10.1145/363534.363546

J. Rudermann. 2001. The Same Origin Policy. Retrieved February 29, 2016 from https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

H. Schorr and W. M. Waite. 1967. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM* 10, 8, 501–506. DOI:http://dx.doi.org/10.1145/363534.363554

Matthew L. Seidl and Benjamin G. Zorn. 1998. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM Press, New York, NY, 12–23. DOI:http://dx.doi.org/10.1145/291069.291012

Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. 2002. Exploiting prolific types for memory management and optimizations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'02)*. ACM Press, New York, NY, 295–306. DOI:http://dx.doi.org/10.1145/503272.503300

Fridtjof Siebert. 2000. Eliminating external fragmentation in a nonmoving garbage collector for Java. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*. ACM, New York, NY, 9–17. DOI:http://dx.doi.org/10.1145/354880.354883

Fridtjof Siebert. 2008. Limits of parallel marking garbage collection. In *Proceedings of the International Symposium on Memory Management (ISMM'08)*. ACM, New York, NY, 21–29. DOI:http://dx.doi.org/10.1145/1375634.1375638

Sunil Soman, Laurent Daynès, and Chandra Krintz. 2006. Task-aware garbage collection in a multi-tasking virtual machine. In *Proceedings of the International Symposium on Memory Management (ISMM'06)*. ACM, New York, NY, 64–73. DOI:http://dx.doi.org/10.1145/1133956.1133965

Sunil Soman, Chandra Krintz, and Laurent Daynès. 2008. MTM2: Scalable memory management for multi-tasking managed runtime environments. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer, Berlin, 335–361.

Steve Souders. 2013. HTTP Archive. (2013). Retrieved February 29, 2016 from http://httparchive.org/trends.php. Accessed 01/10/2013.

Bjarne Steensgaard. 2000. Thread-specific heaps for multi-threaded programs. In *Proceedings of the International Symposium on Memory Management (ISMM'00)*. ACM, New York, NY, 18–24. DOI:http://dx.doi.org/10.1145/362422.362432

Mads Tofte. 1998. A brief introduction to regions. In *Proceedings of the International Symposium on Memory Management (ISMM'98)*. ACM, New York, NY, 186–195. http://doi.acm.org/10.1145/286860.286882.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2, 109–176. DOI:http://dx.doi.org/DOI:10.1006/inco.1996.2613

David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. ACM, New York, NY, 157–167. DOI:http://dx.doi.org/10.1145/800020.808261

Kiem-Phong Vo. 1996. Vmalloc: A general and efficient memory allocator. *Software—Practice and Experience* 26, 3, 357–374. DOI:http://dx.doi.org/10.1002/(SICI)1097-024X(199603)26:3⟨357::AID-SPE15⟩3.0.CO;2-#

Gregor Wagner. 2011. *Domain Specific Memory Management in a Modern Web Browser*. Ph.D. Dissertation. Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA.

Gregor Wagner, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. 2011. Compartmental memory management in a modern web browser. In *Proceedings of the International Symposium on Memory Management (ISMM'11)*. ACM, New York, NY, 119–128.

Paul R. Wilson. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM'92)*. Springer-Verlag, London, UK, 1–42. http://dl.acm.org/citation.cfm?id=664824.

Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. 1991. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*. ACM Press, New York, NY, 177–191. DOI:http://dx.doi.org/10.1145/113445.113461

Benjamin G. Zorn. 1989. *Comparative Performance Evaluation of Garbage Collection Algorithms*. Ph.D. Dissertation. EECS Department, University of California, Berkeley, Berkeley, CA. http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/5313.html.

Received April 2013; revised June 2015; accepted December 2015